2015

Improved Periodicity Mining in Time Series Databases

Nithin Uppalapati

Improved Periodicity Mining in Time Series Databases

Nithin Uppalapati

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Donald Adjeroh, Ph.D., Chair
Elaine Eschen, Ph.D.
Vinod Kulathumani, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2015

Keywords: time series, periodicity mining, suffix tree, compressed suffix tree, approximate periodicity

Abstract

Improved Periodicity Mining in Time Series Databases

Nithin Uppalapati

Time series data represents information about real world phenomena and periodicity mining explores the interesting periodic behavior that is inherent in the data. Periodicity mining has numerous applications such as in weather forecasting, stock market prediction and analysis, pattern recognition, etc. Recently, the suffix tree, a powerful data structure that efficiently solves many strings related problems has been used to gather information about repeated substrings in the text and then perform periodicity mining. However, periodicity mining deals with large amounts of data which makes it difficult to perform mining in main memory due to the space constraints of the suffix tree. Thus, we first propose the use of the Compressed Suffix Tree (CST) for space efficient periodicity mining in very large datasets. Given the time-space trade-off that comes with any practical usage of the CST, we provide a comprehensive empirical analysis on the practical usage of CSTs and traditional suffix trees for periodicity mining.

Noise is an inherent part of practical time series data, and it is important to mine periods in spite of the noise. This leads to the problem of approximate periodicity mining. Existing algorithms have dealt with the noise introduced between the occurrences of the periodic pattern, but not the noise introduced in the structure of the pattern itself. We present a taxonomy for approximate periodicity and then propose an algorithm that performs periodicity mining in the presence of noise introduced simultaneously in both the structure of the pattern and between the periodic occurrences of the pattern.

Acknowledgements

I would like to thank my research advisor Dr. Donald Adjeroh, for the relentless support and guidance he has given me throughout the entire period of my work and through the various phases of my graduate studies. It has been a great pleasure to work with him. I also thank my committee members Dr. Elaine Eschen and Dr. Vinod Kulathumani for their suggestions and corrections. I am thankful to the LCSEE department for providing me with valuable resources and funding through the Graduate Teaching Assistantship. I thank Professor Cindy Tanner for her role as a mentor and guide during my college studies at WVU. I express my deepest gratitude to my parents and my brother who have shown unconditional love and care throughout my life. Am thankful to my friends who have given me the moral support and helped me get through harder times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem

Time series data represents chronological events related to real world phenomena and repeating cycles are an inherent part of the data. Given a time series data, the problem of periodicity mining is to identify all the interesting temporal regularities in the data. What is interesting is often defined by the application, for instance, periodic patterns that are of certain length and have occurred a certain number of times, or patterns that occur periodically within some limited region of the time series data. This requires an understanding of the inherent nature of such cycles, for instance, the structure of the repeating patterns, the noise and uncertainties in such patterns and in the occurrence periods, making effective and efficient periodicity mining a challenging problem. Periodicity mining has interesting applications in various domains, such as, weather prediction, forecasting stock market growth, analyzing patterns of power consumption and many more.

Various approaches have been devised to mine periodicities of different kinds (e.g., symbol periodicity, segment periodicity and sequence or partial periodicity) present in the data. Periodicity mining algorithms either require the user to input the period of interest or mine all the periods that are inherent to the data. Recent algorithms have focussed on the later

case, where the goal is to mine all the possible periods in the data. Prominent work has been done by Indyk et al. [35], Elfeky et al. [22, 23] and others [33, 49, 71, 72]. The recent work by Rasheed et al. [57] utilized the advantages of the suffix tree, a powerful data structure that solves many string related problems efficiently, to capture repetitions in the text. The suffix tree is a data structure that represents all the suffixes of a given text in a compact form and the path label of any given internal node (the concatenation of characters on the path from the root to the given node) represents a unique substring in the text. The list of all the leaf nodes in the subtree rooted at any given internal node stored in an occurrence vector (V), represents the positions in the text of exact occurrences in the given substring. The algorithm proposed by Rasheed et al. [57] called STNR (Suffix Tree based Noise Resilient) algorithm makes multiple scans through the occurrence vector to check whether the current substring is periodic with any period value in different subsections of the text. Algorithm STNR handles the noise introduced between the occurrences of the periodic pattern by considering the occurrences that drift away from the exact periodic positions by a certain value to be part of the periodic occurrences.

Periodicity mining involves the processing of large amounts of data and it is difficult to hold the suffix tree in main memory due to its space constraints. Thus, rather than using suffix trees as in Rasheed et al. [57], we propose the use of Compressed Suffix Trees (CST) for periodicity mining. Even though compressed suffix trees were proposed as a space efficient replacement to suffix tree [51, 60], the practical time for construction and operations defined on the CST could be slower when applied to periodicity mining. Thus, we provide a comprehensive empirical analysis of the practical usage of CSTs and suffix trees for periodicity mining. We use the work by Rasheed et al. [57] as the basis to generate occurrence vectors using CSTs and suffix trees and provide a comparative analysis.

Noise is an inherent part of practical time series data. The time series data is prone to errors from different sources, for instance, data acquisition method, transient errors, inexact periodicity, etc [7]. It is important to mine periods efficiently and effectively in spite of the presence of noise which makes it an even more challenging problem. The existing algorithms [23, 33, 57] have dealt with the noise introduced between the occurrences of the periodic pattern but not the noise introduced in the structure

of the pattern. In this work, we propose Algorithm STNR-A (Suffix Tree based Noise Resilient algorithm with Approximation) to handle the simultaneous case of noise in both the structure of the pattern and between the periodic occurrences. The algorithm employs approximate pattern matching between the periodic pattern and substrings of text surrounding the expected occurrence positions while scanning the occurrence vector. This results in an improved confidence in the periodicity for a given pattern and produces improved accurate mining. However, in the case of long patterns, the time involved in checking the approximate versions of the pattern at missed expected occurrence positions would be very costly. To deal with this problem, we propose the use of a two-phase hypothesis-generation and hypothesis-verification approach using approximate q-gram filtering for practical time improvement. For a given pattern, we generate hypothesis based on exact q-gram matching to filter out regions of text which could be possible regions for approximate occurrences of the pattern. The generated hypotheses are then used in processing the occurrence vector for a given pattern to eliminate certain checks for approximate occurrences.

1.2 Thesis Contributions

The contributions of the thesis are summarized as follows:

1. A comparative analysis on the use of compressed suffix trees and traditional suffix trees for periodicity mining.
2. A taxonomy for approximate periodicity.
3. A new algorithm for periodicity mining that handles noise introduced simultaneously in both the structure of the periodic pattern and between the occurrences of the periodic pattern.
4. A two-phase hypothesis-generation and hypothesis-verification approach for practical improvement in the time required for detecting the periodicities involving long patterns.

1.3 Thesis Outline

In Chapter 2, we present some background and related work on periodicity mining in time series databases and on suffix data structures. We also give a brief introduction to approximate pattern matching. In Chapter 3, we provide a comprehensive empirical analysis of the use of suffix trees and compressed suffix trees for periodicity mining. In Chapter 4, we provide a taxonomy for approximate periodicity and suggest a new algorithm for the most difficult case of approximate periodicity. We also propose an approach with practical time improvement in the case of long patterns. Finally, we provide conclusions in Chapter 5.

Chapter 2

Background and Related Work

2.1 String Pattern Matching

A string T is an ordered sequence of characters $T = t_0t_1.....t_{n-1} = T[0...n-1]$ defined over a fixed alphabet Σ (of size $\sigma = |\Sigma|$). Let $n = |T|$ denote the length of the string and $T[i]$ denote the character at the i^{th} position in the string. An empty string is denoted by ε ($n = 0$). If there exist two strings u and v such that $T = uxv$, then x is called a factor (or substring) of the string T; x is the prefix of T, if $u = \varepsilon$ and x is the suffix of T, if $v = \varepsilon$. The string $T[i...j]$ is a substring that starts at position i and ends at position j. The string $T[0...j]$ is a prefix of the string, which is a substring that starts at position 0 and ends at position j. The string $T[i...n-1]$ is a suffix of the string, which is a substring that starts at position i and continues till the end position $(n-1)$ of the text. There will be a total of n suffixes for a given string denoted by $s_i = T[i.....n-1](0 \leq i \leq n-1)$. In our study, we use \$ ($\notin \Sigma$) as a sentinel character and it is considered to have the lexicographically smallest value when compared with any other symbol in the text T.

The problem of pattern matching is to find all the occurrences of a given pattern P, in the text T. There are two variations to this problem, namely, exact pattern matching and approximate or inexact pattern matching.

2.1.1 Exact Pattern Matching

The problem of exact pattern matching is stated as follows: Given a text T of length n and a pattern P of length m over alphabet Σ, find the starting positions in text T, of all the exact occurrences of the given pattern P. There are several efficient algorithms to solve this problem. The well-known algorithms Boyer-Moore (BM) [15], Knuth-Morris-Pratt (KMP) [40], Apostolico-Giancarlo [9] provide linear worst case time solutions i.e. linear in the size of the text, where the pattern is preprocessed and the text is input online to the algorithm. Other algorithms use interesting data structures such as suffix trees, suffix arrays, and Burrows-Wheeler Transform (BWT) to perform pattern matching, where the text is preprocessed and the patterns of interest are input online. As an example, the text is preprocessed to build the suffix tree in linear space and linear time with respect to the size of the text, and the pattern can be searched in time proportional to the size of the pattern rather than the size of the text [29]. This can be used to efficiently solve the online dictionary search problem, where the text is available and preprocessed and the patterns of interest are searched as they arrive. Other algorithms preprocess both the text and the pattern. The well-known algorithm Karp-Robin [39] transforms both the text and pattern into numerical data called fingerprints using a hash function and then searches for the pattern. The algorithm by Baeza-Yates and Gonnet [11] uses bit-wise operations on binary strings for pattern matching. The books by Gusfield [29], Charras and Lecroq [17], Smyth [63], and Adjeroh et al. [2] provide a detailed discussion of several of these algorithms.

2.1.2 Approximate Pattern Matching

A string T' is said to be an approximate/inexact version of another string T if one can be obtained from the other by performing few transformations on individual characters of either of the strings. A transformation operation can be an insertion, a deletion and a substitution/replacement. Insertions and deletions change the length of the string but replacements do not. The measure of similarity between any two strings is given by the number of such transformations required. One such measure of similarity is the Hamming distance which measures the number of positions at which corresponding

characters are different i.e., the number of substitutions required to transform one string to another. Another distance metric called the edit distance $ED(T, T')$ is the number of edit operations (insertions, deletions, substitutions) required to transform one string to another. We focus on the edit distance for our study.

The problem of approximate pattern matching is stated as follows: Given a text T of length n and a pattern P of length m over alphabet Σ, find all the approximate occurrences of a given pattern in the text i.e., occurrences of the pattern in the text allowing some errors. Typically, the number of errors is bounded to be at most k. Here we take into consideration various types of errors namely replacement, insertion and deletion errors with a bound on the number of errors equal to k. Therefore the problem of inexact matching reduces to that of k-approximate matching, that is finding all the positions i in T, where $ED(S_i, P) \leq k$, where S_i is any substring of T, starting at position i.

The simplest and most popular method to calculate the edit distance between two strings is based on dynamic programming with time complexity of $O(mn)$ [61]. Other methods are based on bitwise operations [53,68] and longest common subsequence [42, 43]. When k, the number of errors allowed is known, the edit distance can be calculated using Ukkonen's [64] approach of deterministic finite state automation in $O(kn)$ time. Navarro [52] provides a comprehensive survey on approximate pattern matching. More detailed discussion on string matching (both exact and inexact pattern matching) is provided in the books [2, 20, 29, 63].

2.2 Repetitions and Periodicity in Strings

The study of repetitions in strings has gained a lot of importance over the years in the field of string combinatorics with numerous applications in areas such as data compression, pattern matching and computational biology. The core of many data compression techniques lies in detecting the repetitions in strings. The study of tandem repeats is important in the field of genome analysis with applications such as disease diagnosis, genomic fingerprinting, etc.

A repeat is an immediately repeated factor or substring in a given string T. There are various types of repetitions that exist in a string such as squares, cubes, and runs (also called maximal repetitions). If a string T is of the form $T = up^e v$, where u and v are non-empty strings, then p^e is called a repetition in the string T, p is called the period, and e the exponent.

When $e = 2$, the repetition is called a square or a tandem repeat and when $e = 3$, the repetition is called a called cube. For example, in string $T = aaabab$, the substring $abab = (ab)^2$ is a square, and $aaa = (a)^3$ is a cube. A tandem repeat is called a primitive tandem repeat if it does not contain shorter repeats. The upper bound on the number of such primitive squares or primitive tandem repeats that can exist in a given string of length n, is $O(n \log n)$, and there exist several algorithms [10,19,47] that compute all the primitive tandem repeats in time $O(n \log n)$.

If there exists a substring $S = T[i...j]$ of string T, that has a period p, such that ($|p| \leq |S|/2$), and periodicity cannot be extended to the right or to the left i.e., $T[i-1] \neq T[i+p-1]$ and $T[j-p+1] \neq T[j+1]$, then the substring representing the interval $[i...j]$ is the called maximal period or run, with an exponent $e = \left\lfloor \frac{|S|}{p} \right\rfloor$. To simply state, a run is a maximal occurrence of a repetition with exponent value ≥ 2 [21].

According to Sim et al. [62], a string P is called a period of string T, if T can be written as $T = P^i P'$, where $i \geq 1$ and P' the prefix of P. And the string T is said to be periodic if $|P| \leq |T/2|$. T is said to be primitive if there exist no periods in it i.e., when $i = 1$. And when $i \geq 2$, then it said to have a repetition P^i.

2.3 Periodicity Mining in Time Series Databases

A time series is a sequence or collection of data gathered over a period of time, generally recorded at uniformly spaced intervals, to analyze the behavior of the object of interest. Time series data represents events related to real-world phenomena such as stock market growth, transactions at a supermarket, hydrological data, power consumption, network traffic, weather data, etc.

Repetitions are an inherent part of the data representing such phenomena. For instance, the ocean tides hit the peaks twice daily, restaurants are busier on certain days during the week. Such repeating patterns reflect the past, present, and possibly future behavior of the phenomenon under observation, and identifying them helps us to understand the nature of the phenomenon and also forecast future events. There are various applications for analyzing such time series, e.g., weather predictions, stock market predictions, pattern recognition, etc. The problem of periodicity detection can be defined as the process of finding the temporal pattern regularities within a time series. The goal of analyzing such series is to detect how frequent, if any, a pattern is repeated in the series [57].

A time series consists of large sequence of symbols or numbers collected at regular intervals gathered over a period of time. Analyzing such a large sequence can be a challenge unless the series is discretized [22, 23, 57]. The time series is discretized by considering an alphabet set (Σ) to represent distinct ranges of values and then mapping the values in the series to the symbol that represents a particular range where the values fall in. For example, consider the time series representing the number of daily visitors to Eiffel tower. To discretize the series, we can consider different possible ranges of visitors; 0 visitors: ***a***, 1-500 visitors: ***b***, 500-1000 visitors: ***c***, >1000 visitors: ***d***. Based on this mapping, the time series $T = 20, 250, 350, 659, 875, 1068, 1564, 450$ can be discretized into T'=***abbccddb***. The discretized time series is then processed in two steps: 1) represent the data in memory, and 2) analyze the data stored [69].

2.3.1 Terminology

Here we formally define the problem and also the terms used in periodicity mining. These definitions are adapted from the papers [22,57]. A time series represents a collection of data gathered at uniformly spaced intervals, given by $T = e_0, e_1, e_2, e_3.....e_n$, where $n = |T|$ represents the length of the time series and e_i represents the value recorded at time stamp i [22,57]. The time series is then discretized using symbols from a predefined alphabet Σ. A periodic pattern (P) is represented using a 5-tuple: $(P, p, stPos, endPos, conf)$, where P is the periodic pattern, p the period value, $stPos$ the position in

the text of the first occurrence of P, $endPos$ the position in the text of the last occurrence of P, and $conf$ represents the degree of confidence on the periodicity [57].

Perfect Periodicity: *A time series T of length n is said to have perfect periodicity with period p, if there exists a pattern P of length m, such that the pattern always occurs in an exact manner from it's first occurrence ($stPos$) till the last occurrence ($endPos$), and the current occurrence is exactly p positions away from its previous occurrence.*

However, perfect periodicity is rarely possible in time series data representing real-world phenomena which leads to imperfect periodicity. The imperfection can be due to the noise introduced in the structure of the pattern which causes certain occurrences to be missed and/or also due to the noise introduced between the occurrences of the pattern which causes the current occurrence to drift by a certain amount away from or towards it's previous occurrence. The later case of noise leads to some local perturbation in the periodic nature of the time series. This is sometimes called an asynchronous periodicity [33, 49, 71, 72]. The degree of perfectness is measured using a parameter called confidence *(conf)*.

Confidence: *The confidence of a periodic pattern P is defined as the ratio of its actual periodicity to its expected perfect periodicity [57] i.e., the actual number of its periodic occurrences to the expected number of occurrences that will make it perfectly periodic.*

For example, consider $T = abcde\ abcce\ abdef\ abeec$, the pattern ab is periodic with $p = 5$, with $stPos = 0$ and $endPos = 15$, and $conf = \frac{4}{4}$ (perfect periodicity). As another example, consider $T = abcde\ abcce\ addef\ abeec$, the pattern ab is periodic with $p = 5$, with $stPos = 0$ and $endPos = 15$, and $conf = \frac{3}{4}$ (imperfect periodicity).

There are three types of periodicity that exist, namely, symbol periodicity, sequence or partial periodicity and segment periodicity [22, 23, 30, 31, 57].

Segment Periodicity [22]: A time series T of length n, is said to have segment periodicity with period p, if it can be split into equal length segments, each of length p, that are *almost* similar. For example, in time series

$T = bcbad\ bcbad\ bcbad\ bcbad$, sequence $P = bcbad$ is periodic with periodicity of $p = 5$, starting from position $stPos = 0$ with perfect periodicity and with $conf = \frac{4}{4}$.

Symbol Periodicity [22]: A time series T of length n, is said to have symbol periodicity with period p, if there exists atleast one symbol that occurs almost every p time-stamps from its first occurrence position. For example, in time series $T = bac\ bdc\ bca\ bde$, symbol $P = b$ is periodic with periodicity of $p = 3$, starting from position $stPos = 0$ and with $conf = \frac{4}{4}$.

Sequence or Partial Periodicity: A time series T of length n, is said to have sequence or partial periodicity with period p, if there exist a sequence of symbols that occur *almost* every p positions from their first occurrence. For example, in time series $T = acbb\ acdb\ aced\ acdb$, sequence $ac**$ is periodic with a periodicity of $p = 4$, starting from position $stPos = 0$, and $conf = \frac{4}{4}$ where $*$ denotes a don't care symbol. Also, another sequence $ac*b$ is periodic with a periodicity of $p = 4$, starting from position $stPos = 0$ and $conf = \frac{3}{4}$.

2.3.2 Periodicity Detection Methods

There are several algorithms for time series analysis and periodicity detection. The algorithms can be divided into two categories, ones that require the user to input the period of interest and search for patterns repeating with that specific behavior, and the others that look for all possible periods in the series. The algorithms can also be categorized based on the type of periodicity they detect, and also whether they detect periods in the whole time series or only in a subsection of the time series. The algorithms in [13,31,46,70] require the user to input the period of interest, however, they do not fulfill the goal of mining unexpected periods. The algorithm ParPer by Han et al. [30] detects only partial periodic patterns in linear time provided the expected period value is given. The time complexity increases to $O(n^2)$ to detect all possible periods. The goal of recent periodicity mining algorithms is to detect all the possible periods in the data. Indyk et al. [35] presented an $O(n \log^2 n)$ time algorithm to detect segment periodicity. This was later improved by Elfeky et al. [22] to detect segment and symbol periodicity in $O(n \log n)$ time by using convolution based technique (CONV). Noise is inherent in time series

data and the CONV algorithm fails to deal with insertion and deletion noise. A second algorithm was developed by Elfeky et al. [23] referred to as WARP (time warping for periodicity detection) which detects segment periodicity in the presence of insertion and deletion noise with a time complexity of $O(n^2)$. The algorithms CONV and WARP cannot detect patterns that are periodic only in a subsection of time series. The algorithms E-MAP (Efficiently Mining Asynchronous Periodic patterns) [49], OEOP (One Event One Pattern mining) [72], LSI (Longest Subsequence Identification) [71], SMCA (Simple Multiple Complex and Asynchronous periodic pattern miner) [33] efficiently mine asynchronous periodic patterns where the expected periodic occurrences are allowed to drift from their expected positions up to an allowable limit due to the noise introduced between the occurrences of the periodic pattern.

The algorithm by Rasheed et al. [57] can be used to efficiently detect the three types of periodicity, namely, segment, symbol and partial periodicities. It can also detect periodicity starting from all positions in the whole or in a subsection of time series data with the time complexity of $O(n^3)$. They used suffix trees to efficiently capture repeating substrings of the time series data, where the path label of the internal nodes of the suffix tree represents the repeated substrings in the data. The starting positions of all the occurrences of a given substring are recorded in an occurrence vector V. This is used as an input to the periodicity mining algorithm. The algorithm handles insertion and deletion noise to efficiently mine asynchronous periodic patterns.

In summary, the algorithm by Rasheed et al. [57] uses the suffix tree which is a powerful data structure that solves many strings related problems. However, the suffix tree cannot be used for large data sets because of its large memory footprint. Their algorithm also fails to deal with the noise introduced in the structure of the periodic pattern. Therefore, we first suggest the use of CSTs for space-efficient periodicity mining. Then, we provide an empirical analysis on the use of CSTs as a replacement to suffix trees for periodicity mining and use various data sets that contain repetitive texts for analysis. We also provide a solution that efficiently mines periodic patterns in the presence of noise simultaneously both in the structure of the periodic pattern and also between the occurrences of the periodic pattern.

2.4 Suffix Trees and Suffix Arrays

2.4.1 Suffix Trees

A suffix tree (ST) is a data structure that provides a compact representation of all the suffixes in a given string. It can be used to perform important operations on strings in an efficient manner and has a wide range of applications such as exact matching, search for a regular expression, longest common substrings, lowest common ancestor, and many other as mentioned in Dan Gusfield's text [29]. According to Gusfield [29] a suffix tree constructed for a text of length n (including the sentinel character \$) has the following properties:

- Is a rooted directed tree with exactly n leaves numbered 0 to $n-1$.
- Each internal node other than the root has at least two children.
- Each edge is labelled with a non-empty substring of T
- No two edges out of a node can have edge-labels beginning with the same symbol.
- The concatenation of the edge labels on the path from the root to a $leaf-i$ represents the i^{th} suffix $T[i....n-1]$.

The path label of a node is the concatenation of all the symbols from the root to the given node. Since no two edges out of a given node can have edge labels beginning with the same symbol, every internal node's path label represents a unique substring of the given text T. The key point is that the internal nodes efficiently capture the repetitions in the text and all the occurrence positions of a given substring (path label of the internal node) in the text are given by the list of leaf node numbers in the subtree rooted at that given internal node. The average depth of a suffix tree is $\log n$ and the upper bound on the number of nodes in a suffix tree is $2n-1$ with always n leaves each representing a different suffix of the text and at most $n-1$ internal nodes. Figure 2.1 shows an example suffix tree for the sequence T=***mississippi***\$.

The naive approach to construct a suffix tree takes time proportional to $O(n^2)$. Wiener [67] proposed the first linear time algorithm to construct a suffix tree that was later improved with a simplified space efficient construction method by McCreight [50]. A simpler algorithm for linear time construction of the suffix tree was proposed by Esko Ukkonen [65] and is widely used for its simple implementation details and space efficiency and also for its online property. The use of suffix links and skip/count trick makes it [65] a linear time algorithm for suffix tree construction. According to Ukkonen [65], if $x\alpha$ (where x is a single character) is the path label of an internal node v and if there is an another internal node $s(v)$ with path label α, then the pointer from node v to $s(v)$ is the called the suffix link for node v and the pointer from v goes to the root if α is an empty string (ε). It is observed that all the internal nodes have suffix links to other internal nodes and the use of suffix links can solve certain string analysis problems more efficiently, e.g., circular pattern matching [44] and circular pattern discovery [45].

The space required to store a suffix tree will include that of the original text T, edge labels, node labels for both branching and leaf nodes, the space to indicate the parent for each node and the suffix links [2]. It requires n bytes for the text, $2n$ integer pointers for edge labels (edge labels are represented by a pair of indices that specify the beginning and end positions of a substring of the text), $2n$ integer pointers for node parents, $2n$ integer pointers for node labels for both branching and leaf nodes and $2n$ integer pointers for suffix links thus giving us a total of $33n$ bytes to store the suffix tree, where a pointer is represented using an integer which is 4 bytes long and 1 byte is used to represent each character of the text. The theoretical space consumed by a suffix tree is $O(n \log n)$ bits as there are $O(n)$ pointers and $(\log n)$ bits is required to encode each pointer.

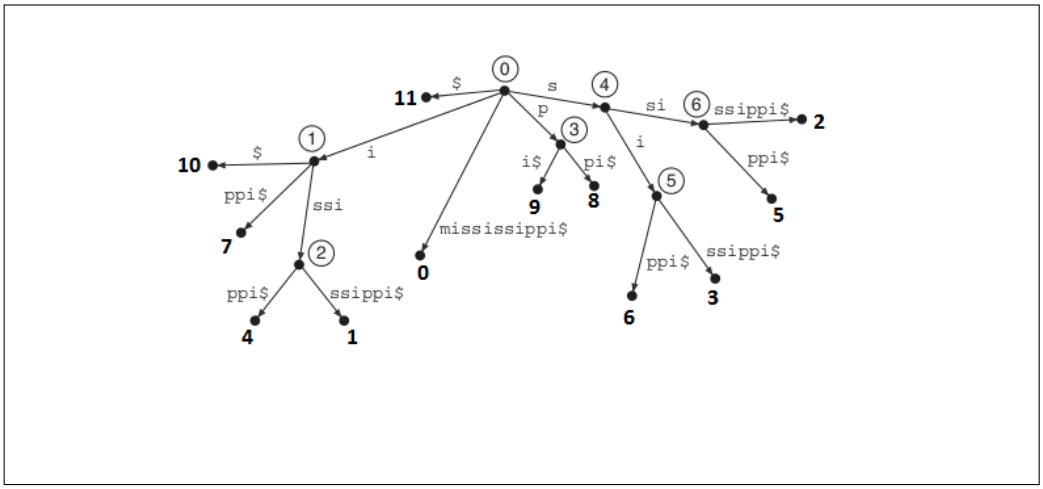


Figure 2.1: Suffix tree structure for T=***mississippi***\$**.**

In Figure 2.1, the internal nodes are labelled in the order the nodes are visited during a depth first search (dfs) traversal of the tree structure. The leaf nodes are marked with the corresponding suffix numbers representing the node's path label.

2.4.2 Suffix Arrays

The suffix array for a given text T (of length $n = |T|$) is an array of indices $SA[0...(n-1)]$ that represents the lexicographically sorted order of all the suffixes of the text. That is, the starting position in the text of the lexicographically smallest suffix is stored at $SA[0]$ and starting position of the lexicographically second smallest suffix is stored at $SA[1]$ and so on, such that $SA_T[0] < SA_T[1] < ..... < SA_T[n-1]$, and $SA_T[i]$ represents the suffix starting at index $SA[i]$ in the text ($0 \leq i \leq n-1$). Suffix arrays were introduced as a space efficient replacement for suffix trees. A suffix array can be generated in linear time by recording the leaf numbers through a depth-first traversal of a suffix tree provided the edges at each internal node are lexicographically ordered. However, due to the space consumption constraints of suffix trees, new approaches to directly construct suffix arrays have been pro-

posed. Manber and Myers [48] introduced the first algorithm to construct suffix arrays which occupies only $O(n \log n)$ bits of space with an average time of $O(n)$ and worst case time of $O(n \log n)$. Other linear time algorithms include those proposed by Kärkkäinen and Sanders [38], Ko and Aluru [41], Adjeroh and Nan [3]. The suffix arrays can be used to search for a pattern P of length $m = |P|$ in a text of length n in $O(m \log n)$ time as it requires $O(\log n)$ comparisons to perform a binary search on a suffix array and in each comparison P is compared with a suffix to determine the lexicographic order which at most takes $|P|$ comparisons [5]. Although the suffix arrays were used as a replacement to suffix trees due to space constraints in storing the suffix tree data structure, the actual space required by the suffix sorting algorithms during the construction of suffix arrays is very large. Also, the suffix arrays do not store information about suffix links which could solve certain problems efficiently. A suffix tree can be constructed from a suffix array with the help of an auxiliary array called the LCP (Longest Common Prefix) array. And the search for a pattern can be reduced to $O(m + \log n)$ with the help of this auxiliary LCP array.

The Longest Common Prefix (LCP) array is an array data structure which holds the length of the longest common prefix between the consecutive suffixes for the set of all the lexicographically sorted suffixes of a string. In a way it stores the LCP values of the suffixes represented by consecutive elements of a suffix array i.e., $LCP[0] = -1$ and $LCP[1]$ = length of the longest common prefix between suffixes $SA[0]$ and $SA[1]$ and so on.

$$LCP = \begin{cases} -1 & i \in \{0, n\} \\ |lcp(T_{SA[i-1]}, T_{SA[i]})| & otherwise \end{cases}$$

i	$SA[i]$	$LCP[i]$	T_i
0	11	-1	$
1	10	0	i$
2	7	1	ippi$
3	4	1	issippi$
4	1	4	ississippi$
5	0	0	mississippi$
6	9	0	pi$
7	8	1	ppi$
8	6	0	sippi$
9	3	2	sissippi$
10	5	1	ssippi$
10	2	3	ssissippi$

Table 2.1: Information on SA and LCP for the sequence T=***mississippi***$\$$**.**

2.5 Compressed Suffix Trees and Compressed Suffix Arrays

2.5.1 Succinct Data Structures

Jacobson [36,37] was among the first to introduce the concept of succinct data structures to represent a particular data structure in its compressed format with the amount of space close to the information-theoretic lower bounds, and yet the operations defined on it be performed efficiently, without the need for decompression. It is observed that, some of the operations on these succinct data structures can be performed with the same time complexities as that of its uncompressed counterparts.

The suffix tree is efficient in solving many strings related problems for applications such as genomic sequence analysis, data mining, bioinformatics, information retrieval etc. The major drawback with the use of suffix trees for such applications that require processing large data is their very large space requirements and also the necessity to hold the data structure in main memory [58]. For example, a space efficient suffix tree takes at least 40 Gigabytes of main memory for the human genome sequence of size 700 Megabytes [60]. It is stated that, in theory the suffix tree occupies $O(n \log n)$ bits of space to store the data structure, but the constant involved in the $O(.)$ notation is significant in practice [25] and it depends on the implementation methods used. As stated earlier, a typical implementation of suffix tree requires the space of about 33 times the size of the text and at least 25 times the size of the text without storing the suffix links (where a pointer is represented using an integer of size 4 bytes). The practical space requirements will be even more for systems with 64-bit architecture [25]. One way to deal with this problem is the use of succinct data structure called the Compressed Suffix Tree (CST) and the goal is to reduce the space requirements from $O(n \log n)$ bits to $O(n \log |\Sigma|)$ bits or even $O(n)$ bits and perform operations on the compressed format with similar time complexities as compared to its uncompressed counterpart [2].

2.5.2 Compressed Suffix Arrays

Compressed suffix array (CSA) was proposed to replace the suffix array in space, but surprisingly they provide more functionality than the uncompressed suffix array [25, 58]. The size of a suffix array can be reduced from $(n \log n)$ bits to $O(n \log |\Sigma|) + o(n \log |\Sigma|)$, (where $|\Sigma|$ represents the size of the alphabet) with the proposal by Grossi and Vitter [28], but with an increase in the access time from constant time to $O(\log^{\epsilon} n)$, (where ϵ is any constant such that $0 < \epsilon \leq 1$). They also proposed the construction of compressed suffix trees from compressed suffix arrays. CSA proposal by Sadakane [59] is called a self indexing structure as it replaces the original text i.e., it can provide fast access to the string without explicitly storing the text string T [25]. The space required for Sadakane's proposal [59, 60] of CSA is $O(n \log H_0 + n \log \log |\Sigma|)$, where H_0 is the zero-th order entropy for the text T. The access time is increased from constant time to $O(\log^{\epsilon} n)$ [59, 60], similar to the proposal by Grossi and Vitter [28]. Navarro and Mäkinen [54] present a survey on various proposals for efficient construction of the compressed suffix arrays.

There exists a one-to-one correspondence between the symbols of text T of length n and the elements of suffix arrays which gives the motivation for compression [2]. Grossi and Vitter [28] proposed the solution by representing the suffix array as an abstract data type [2], with the help of two operations defined for a text T of length n and suffix array SA as follows:

- *compress(T,SA)*: this operation compresses the suffix array SA to generate compressed suffix array (CSA). The CSA thus generated is retained along with the text T, while discarding the original suffix array.
- *lookup(i)*: this operation returns the index position in the suffix array of the lexicographically i-th smallest suffix in T.

The goal is to develop a compressed suffix array in reduced space and yet provide operations comparable to those of the uncompressed counterpart. The *lookup(i)* operation can be performed in constant time with a compressed suffix array. It was shown by Grossi and Vitter [28] that the operation *lookup(i)* can be performed in $O(\log \log n)$ time with a CSA represented using $\frac{1}{2} \log \log n + 6n + O(\frac{n}{\log \log n})$ bits of space. Similarly, for general

alphabet $|\Sigma| > 2$, the operation *lookup(i)* can be performed in $O(\log \log_{|\Sigma|} n)$ time with a CSA represented using $1 + \frac{1}{2} \log \log_{|\Sigma|} n + 5n + O(\frac{n}{\log \log n})$ bits of space [2].

2.5.3 Compressed Suffix Trees

Compressed Suffix Trees (CST) were proposed as a replacement to Suffix Trees (ST) to represent them in compact space and have the defined operations still be performed efficiently. A compressed suffix tree consists of the following three parts, each of which is represented in a compressed form.

1. A compressed suffix array(CSA): self-indexing structure which provides lexicographical information. It is called a self indexing structure as it can replace the text.
2. A compressed LCP Array: provides information about common substrings in the text.
3. Navigation structure (NAV): a succinct data structure which represents the tree topology and supports navigational operations efficiently.

Thus, the size of the CST is given by:

$$|\text{CST}| = |\text{CSA}| + |\text{LCP}| + |\text{NAV}|, \text{ where } || \text{ represents the size.}$$

Different combinations of these components lead to different variants which have been studied by different authors. Compressed suffix trees were originally introduced by Munro et al. [51] which represents the suffix tree in $n \log n + O(n)$ bits of space and allows the search for a pattern in $O(m)$ time. Sadakane [60] proposed a different version of CST that is linear in size of the text and occupies $nH_0 + O(n \log \log |\Sigma|) + 6n + o(n)$ bits of space where the LCP parts takes $2n + o(n)$ bits and NAV part takes $4n + o(n)$ bits. The NAV part of Sadakane's CST is represented using a balanced parenthesis sequence ($4n$ bits) with a support structure of sub-linear space $o(n)$ bits. The first practical implementation of CST was given by Välimäki et

al. [66] based on the proposal by Sadakane [60]. The CST proposal by Russo et al. [58] occupies sub-linear space, but the runtime operations are slower compared to the Sadakane's [60] version of CST. The proposal by Fischer et al. [24] occupies less space as compared to Sadakane's version of CST. They used a run length encoded version of Sadakane's LCP and the NAV part is dependent on the operations of LCP. The proposal by Fischer et al. is implemented by Cȧnovas and Navarro [16] and the operations are orders of magnitude slower in practice. Recently, Gog et al. provided a library of CST implementations [25, 26] based on their theoritical proposals in [27, 55, 56], with two efficient implementations of CST. The first is CST_SADA which is an optimized implementation of Sadakane's proposal [60] in which the nodes are represented based on its position in the BPS (balanced parenthesis sequence) and the navigational operations are faster. The second is CST_SCT3 based on their proposal in [55] that requires $3n + o(n)$ bits to represent the NAV. The NAV part of CST_SCT3 does not depend on the LCP for its operations and the navigational operations are slower as compared to the CST_SADA version of CST.

Chapter 3

Periodicity Mining using CSTs

3.1 Introduction

Compressed Suffix Trees (CST) were designed as a replacement for Suffix Trees (ST) with the goal to represent them in a compact space, while performing the operations on suffix trees efficiently. However, with any implementation of the CST there is a time-space trade-off which leads to a slow down of the operations by some factor. The use of CST rather than ST is the choice to be made depending upon several factors such as the size of the text, application needs, and the operations that need to be performed. Periodicity mining on time series databases deal with large amounts of data which makes it difficult to perform mining in main memory due to the space requirements of the suffix tree. Here, we provide an empirical analysis on practical usage of CSTs for periodicity mining. We use the algorithm by Rasheed et al. [57] as the basis, which use suffix trees for periodicity mining in time series databases. Once a suffix tree is constructed for the text representing the time series data, the tree is traversed to record the occurrence vector at each internal node in a bottom-up manner. An occurrence vector records the list of all the leaf nodes in the subtree rooted at any given node. The key idea is that the repetitions of substrings in the text are captured effectively by the internal nodes in the suffix tree and the positions of all the exact occurrences of the substring represented by an internal node's path la-

bel is given by the list of leaf nodes in the subtree rooted at the given internal node. The periodicity algorithm then analyzes the occurrence vector of each internal node to check if the substring representing the internal node's path label is periodic in the given text.

3.2 Using CSTs for periodicity mining

Compressed suffix trees can be used for periodicity mining involving huge amounts of data. Once a compressed suffix tree is built on the text representing the practical time series data, it can be used to generate occurrence vectors for each internal node of the suffix tree data structure. As mentioned earlier, the repetitive substrings in the text are captured efficiently by the internal nodes of the suffix tree data structure and we can efficiently generate the occurrence vectors by traversing the compressed suffix tree with the use of operations given by the navigation structure (NAV) of the CST. Sadakane's [60] proposal for CST uses a balanced parenthesis sequence to represent the tree topology of a suffix tree data structure and with the help of a sub-linear support structure, it provides constant time navigational operations that aid in an efficient traversal of the tree.

Balanced Parenthesis Sequence: The suffix tree data structure can be represented using a nested string of balanced parentheses [2, 51]. Figure 3.1, gives the parentheses sequence and the suffix tree structure for an example string, T=***mississippi***\$. Each node is represented as a pair of open and closed parenthesis [2] and the node is represented by the position of the open and closed parentheses in the string sequence. The balanced parentheses sequence (BPS) can be obtained by performing an inorder traversal on the suffix tree starting from the root, where an open parenthesis is written for an internal node the first time it is visited and then the subtree of the node is visited and a closed parenthesis is written, after all the nodes in the subtree are visited. A leaf node is represented by a pair of consecutive open and closed parenthesis and the operation $is\_leaf(v)$ can be answered in constant time with use of query BPS$[v+1] = 0$, where v represents any node and the value 0 represents a closed parenthesis and 1 represents an open parenthesis in the parentheses notation [25]. For a text of length n, the suffix tree can have at most $2n-1$ nodes, with n leaves and at most $n-1$ internal

nodes. Thus, it requires $4n$ bits to represent the tree topology using the BPS notation.

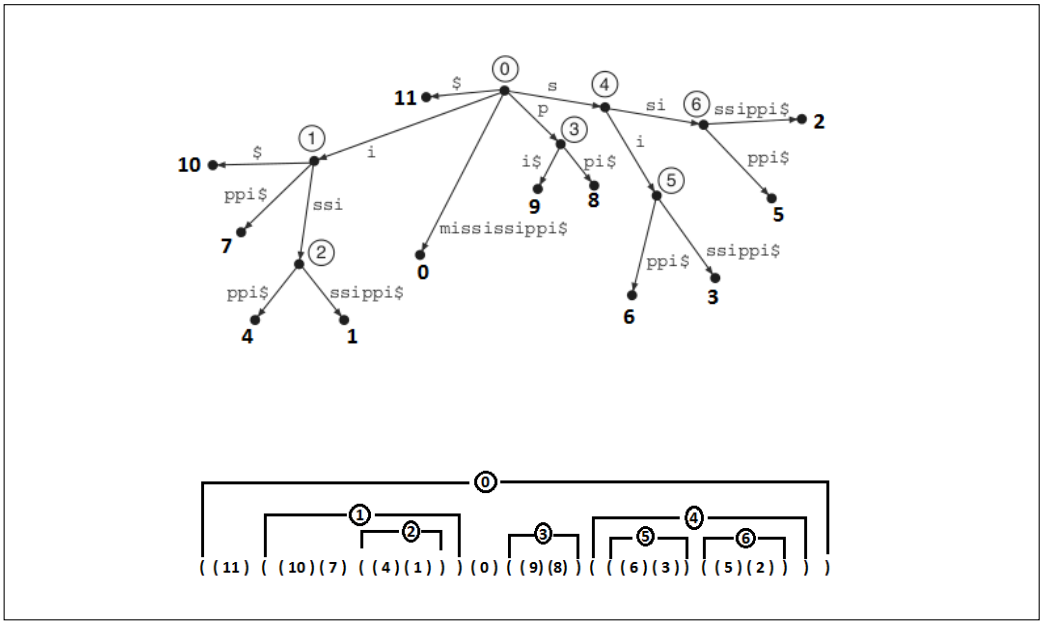


Figure 3.1: Suffix tree structure and BPS notation for sequence T=***mississippi***\$**.**

The operations that aid in navigation of the tree are based on the following two constant-time operations supported by the use of additional supporting data structures that occupy $o(n)$ bits of space:

- *rank(i)*: returns the count of the number of 1's in a prefix BPS[0.....$i-1$] of the binary BPS sequence.
- *select(i)*: returns the position of the i-th 1 in the binary BPS sequence.

The data structures to support these operations are computed in linear time [51]. The following constant time operations use the *rank(i)* and *select(i)* operations to perform efficient navigation [25, 26]:

- *parent*(v): returns the parent node of v and returns v, if $v = root()$

- $sibling(v)$: returns the next sibling of v and returns $root()$, if v is the right most child of $parent(v)$
- $ith\_child(v, i)$: returns the i-th child of the node v for $i \in [1....degree(v)]$
- $leftmost\_leaf\_in\_the\_subtree(v)$: returns the left most leaf in the subtree rooted at node v

Algorithm 1 below shows the pseudo code for traversal of a tree (ST and CST) to generate occurrence vectors for each internal node.

Algorithm 1 Non-recurvise traversal algorithm to generate occurrence vectors.

```
1: Initialize a vector occVec that holds the occurrence vectors for all the
   internal nodes
2: Set index_occVec ← 0
3: Initialize the stack
4: push the root onto the stack
5: while the stack is not empty do
6:     node(v) ← pop the element from the stack
7:     if node(v) is already visited then
8:         if node(v) is not a root node and not a leaf node then
9:             node(v).endIndex ← index_occVec
10:        end if
11:    else
12:        if if node(v) is a leaf node then
13:            add the suffix number of the leaf node to the vector occVec
14:            increment index_occVec
15:        else
16:            mark the node(v) as visited
17:            node(v).startIndex ← index_occVec
18:            push node(v) onto the stack
19:            for each child node(c) of node(v) do
20:                push node(c) onto the stack
21:            end for
22:        end if
23:    end if
24: end while
```

The compressed suffix tree is traversed using a non-recursive post order traversal method. A single vector *occVec* is used to store the occurrence vectors for all the internal nodes in the tree and for a given internal node its occurrence vector is given by making pointers *node.startIndex* and *node.endIndex* into the vector *occVec*. The traversal starts with the root node and if the given node is not already visited, then: if it is a leaf node, the suffix number of the leaf node is added to the vector *occVec*, and if it is an internal node it is marked as visited and the current size of the *occVec* is stored as the *node.startIndex* for the given node and the same is repeated for all the nodes in its subtree. If the node is already visited, then all the nodes in subtree are assumed to be traversed and if it is an internal node, then the current size of the *occVec* is stored as the *node.endIndex* for the given node. Algorithm 2 (page 28) shows, how the DFS iterator defined in sdsl library [25] is used to generate the occurrence vectors using a CST.

3.3 Empirical Comparative Analysis: ST vs CST

3.3.1 Goals of the Empirical Analysis

Our goals in the empirical analysis are as follows:

- Compare the practical size of the suffix data structures.
- Compare the time required to construct the suffix data structures.
- Compare the time required to generate occurrence vectors as described by Rasheed et al. in [57] on which the periodicity algorithm is run.
- Study how various attributes of the sequence influence the performance of a given suffix data structure (ST or CST).

3.3.2 Experimental Setup

Implementation Details

Suffix Tree: We have used an ANSI C implementation by Yona Shlomo and Tsadok Dotan [73]. In this implementation, a node is represented using a C struct with the following members:

Member	Description	Size (bytes)
children	A pointer to linked list of child nodes	8
right_sibling	A pointer to linked list of right siblings of that node	8
left_sibling	A pointer to linked list of left siblings of that node	8
parent	A pointer to that node's parent	8
suffix_link	A pointer to the node that represents the largest suffix of the current node	8
path_ position	Index of the start position of the node's path	8
edge_label_start	Start index of the incoming edge	8
edge_label_end	End index of the incoming edge	8
node_index	Unique identification for the node (Added to help in traversal of the tree)	8
Total		72

Table 3.1: Attributes of a node in the ANSI C Implementation of the suffix tree [73].

This implementation gives efficient construction of suffix trees for the text of any size and any alphabet based on Ukkonen's [65] method for online construction of suffix trees in linear time and linear space. A non-recursive post order traversal with explicit stack-based algorithm [4] shown in Algorithm 1 (page 24) is used to generate occurrence vectors. The method of construction of suffix tree and the method of generation of occurrence vectors is similar to the methods used by Rasheed et al. [57].

Compressed Suffix Trees: We have used the C++ library *sdsl* (Succinct

Data Structure Library) by Simon Gog et al. [26] for compressed suffix trees. We have used the following two variants of the CST implementations:

- CST_SADA: An implementation of Sadakane's [60] proposal for CST where a node is represented by its position in the BPS (Balanced Parenthesis Sequence). The navigational operations are fast and are implemented as BPS operations on DFS-BPS. The worst case space complexity is $|CSA| + |LCP| + 4n + o(n)$ bits, where it takes 2 bits to represent each node using BPS and in the worst case there are $2n$ nodes. The entropy compressed version of CSA occupies $nH_k + |\Sigma|^k$ bits of space and LCP occupies $2n + o(n)$ bits of space, where H_k is the k-th order empirical entropy of the text T.

  $$\text{The } k\text{-th order entropy for text } T\text{: } H_k = \frac{1}{n} \sum_{\omega \in \Sigma^k} |T_\omega| H_0(T_\omega)$$

  $$\text{The zero-th order entropy for text } T\text{: } H_0 = \sum_{i=0}^{|\Sigma|-1} \frac{n_i}{n} \log \frac{n}{n_i}$$

  where T_ω is the concatenation of all the characters in T which follow the occurrences of the substring ω in T [25], and n_i is the number of occurrences of character c_i in T (where $\Sigma = \{c_0, ...c_{|\Sigma-1|}\}$).

- CST_SCT3: An implementation by Gog et al. [55] where nodes are represented as intervals. The construction is fast, but the operations are slower. The worst case space complexity is $|CSA| + |LCP| + 3n + o(n)$ bits. Again, the entropy compressed version of CSA occupies $nH_k + |\Sigma|^k$ bits of space and LCP occupies $2n + o(n)$ bits of space, where H_k is the k-th order empirical entropy of the text T.

The occurrence vectors are generated using the DFS iterator given in the library with slight modifications to store values for the occurrence vectors. The iterator defined takes constant time for increments as it uses constant time operations such as $parent(v)$, $sibling(v)$, $ith\_child(v, i)$, and $leftmost\_leaf\_in\_the\_subtree(v)$ defined in Section 3.2. In addition to these methods, we use the method $sn(v)$ defined in the library, which returns the suffix number of a given leaf node v. The method $sn(v)$ queries the underlying CSA structure to count the leaves to the left of the leaf node

v, with a time complexity of the order of access to suffix array element. Algorithm 2 presents our pseudo code for generating occurrence vectors using the dfs iterator defined in the sdsl library [25].

Algorithm 2 Algorithm to generate occurrence vectors using the dfs iterator defined in the sdsl library for CSTs.

```
1: Initialize a vector occVec which holds the occurrence vectors for all the
   internal nodes
2: Set index_occVec ← 0
3: Initialize the iterator(it) on CST
4: for it.begin() till it.end() do
5:     node(v) ← it.node()
6:     if if node(v) is a leaf node then
7:         add the suffix number to the vector occVec
8:         increment index_occVec
9:     else
10:        if node(v) is not a root node then
11:            if node(v) is not already visited then
12:                node(v).startIndex ← index_occVec
13:            else
14:                node(v).endIndex ← index_occVec
15:            end if
16:        end if
17:    end if
18: end for
```

Simulation Environment

Tests for the analysis were run on a memory-optimized cloud instance picked from Amazon Elastic Cloud Compute (EC2) platform with 30.5 GB of DDR3 DRAM to support memory-intensive computations and with 4 virtual CPUs of type Intel Xeon E5-2670 v2 (Ivy Bridge, 64-bit) processor with 25 MB cache running at 2.50 GHz. An 80 GB SSD storage was configured to hold the programs, scripts, test input datasets and the generated outputs. Ubuntu Server 14.04 LTS (HVM) was the operating system running on this system. The g++ version 4.8.2 (Ubuntu 4.8.2-19 ubuntu1) was used for compilations.

cmake (2.8.12.2) was chosen to automate all the build tasks.

3.3.3 Data Sets

Different kinds of data both synthetic and real data containing repetitions were used in the experiments.

- **Fibonacci Words:** These represent data generated artificially with highly repetitive substrings. The Fibonacci sequence (also called Sturmian sequence) is given by the recurrence relation: $F_1 = 1, F_2 = 0, F_n = F_{n-1} + F_{n-2} \; \forall \; (n > 2)$, where "+" denotes the concatenation of two words. Example sequences for $F_n, n = 1, 2, ...7$: 1, 0, 01, 010, 01001, 01001010, 0100101001001. Although the infinite Fibonacci word is not periodic, it is considered to be extremely repetitive and is often considered as the worst case for algorithms which detect repetitions in a string [14, 34]. A Fibonacci word of length n has $O(n \log n)$ repetitions [6].
- **Real Data:** The real data composed of highly repetitive texts from different sources such as Wikipedia, DNA and documents (information about CIA world leaders) obtained from the *Pizza & Chili* Corpus [18] available at *http://pizzachili.dcc.uchile.cl/repcorpus.html*. The files used are *einstein.de.txt, Escherichia_Coli, world_leaders*.
- **Pseudo Real Data:** These represent the data generated artificially by adding repetitiveness to real data sets, which are also obtained from the *Pizza & Chili* Corpus [18]. Each of these files are of size 100MB.
- **Synthetic Data:** The synthetic data has been generated in the same manner as generated by Elfeky et al. in [22]. A partial periodic data of size 100MB is generated with a set of parameters (dist=uniform, p =32, $|\Sigma|$=10), with similar values as used in [57]. And chunks of varying data sizes of the generated data file are used. Experiments produce similar results for data with normal distribution.

We characterize the data sets using their average LCP, maximum LCP values and their entropy (H_0, and H_1). Considered with the sequence

length (n), these values provide an idea of the difficulty in searching for patterns (including periodicity). Larger values of LCP implies more difficulty. See Table 3.2 for Fibonacci words, Table 3.9 for real data, Table 3.16 for pseudo real data, and Table 3.23 for synthetic data.

3.4 Results

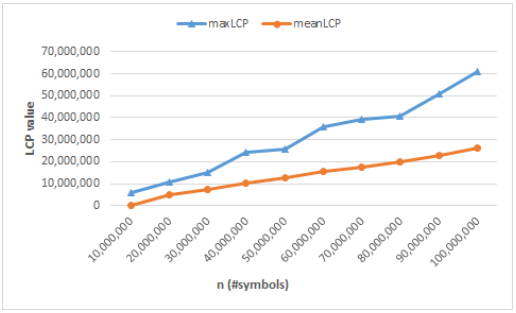
Results reported in this section are averages for 5 repetitions of the same experiment. We study the impact of how various attributes of the sequence influence the performance of a given suffix data structure using correlation tables. See Tables 3.3, 3.4, 3.5 for Fibonacci words, Tables 3.10, 3.11, 3.12 for real data, Tables 3.17, 3.18, 3.19 for pseudo real data, and Tables 3.24, 3.25, 3.26 for synthetic data. The upper half of the tables give Pearson's correlation coefficient values and the lower half of the tables give Kendall Tau rank coefficient values.

3.4.1 Fibonacci Words

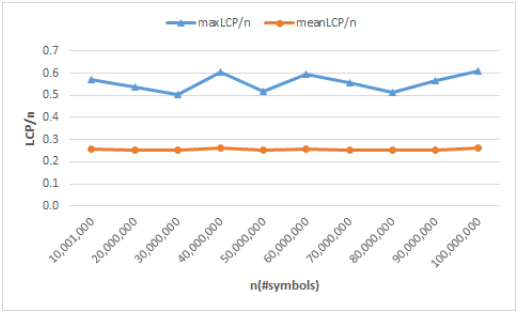
Table 3.2 shows the LCP and entropy information for the data set with Fibonacci words. As expected, the maximum LCP is relatively high, always $(n/2)$) for each given n. The mean LCP is also relatively high, growing very fast with increasing values of n. The per symbol value of max and mean LCP is approximately equal for different data sizes, but does not show any linear behavior with increasing data size. The H_0 and H_1 values are generally constant at 0.959 and 0.593 respectively, independent of the sequence length n. The $(\#numNodes/n)$ is 2 for Fibonacci words.

n (#symbols)	#numNodes	maxLCP	meanLCP	maxLCP/n	meanLCP/n
10,000	20,000	5,819	2,566	0.582	0.257
100,000	199,999	53,632	25,131	0.536	0.251
1,000,000	1,999,997	514,227	250,202	0.514	0.250
10,000,000	19,999,997	5,702,885	2,549,580	0.570	0.255
20,000,000	39,999,913	10,772,535	5,029,840	0.539	0.251
30,000,000	59,999,994	15,069,648	7,500,160	0.502	0.250
40,000,000	79,999,999	24,157,815	10,432,200	0.604	0.261
50,000,000	99,999,999	25,842,183	12,514,200	0.517	0.250
60,000,000	120,000,000	35,842,183	15,568,900	0.597	0.259
70,000,000	140,000,000	39,088,167	17,738,800	0.558	0.253
80,000,000	160,000,000	40,911,831	20,010,400	0.511	0.250
90,000,000	179,999,997	50,911,831	22,888,300	0.566	0.254
100,000,000	199,999,997	60,911,831	26,190,700	0.609	0.262

Table 3.2: LCP, entropy and other attributes for Fibonacci words.



(a) meanLCP and maxLCP



(b) meanLCP/n and maxLCP/n

Figure 3.2: LCP information for varying data sizes(n) using Fibonacci words.

CST_SADA								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	0.895	-0.195	-0.239	-0.859	-0.863	0.481	-0.371
H_1	0.809	1.000	0.030	-0.089	-0.679	-0.770	0.313	-0.420
maxLCP	-0.218	0.036	1.000	0.959	0.193	0.241	-0.410	0.177
meanLCP	-0.161	0.037	0.966	1.000	0.173	0.283	-0.416	0.276
size	-0.723	-0.548	0.090	0.070	1.000	0.913	-0.519	0.323
cTime	-0.715	-0.615	0.156	0.138	0.584	1.000	-0.572	0.487
tTime	0.591	0.470	-0.200	-0.184	-0.494	-0.511	1.000	-0.881
# nodes	-0.676	-0.561	0.000	-0.024	0.442	0.460	-0.506	1.000

Table 3.3: Correlation values for CST_SADA using Fibonacci words.

CST_SCT3								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	0.895	-0.195	-0.239	-0.891	-0.826	0.488	-0.371
H_1	0.809	1.000	0.030	-0.089	-0.724	-0.744	0.337	-0.420
maxLCP	-0.218	0.036	1.000	0.959	0.278	0.195	-0.408	0.177
meanLCP	-0.161	0.037	0.966	1.000	0.285	0.216	-0.419	0.276
size	-0.715	-0.615	0.156	0.138	1.000	0.956	-0.676	0.510
cTime	-0.715	-0.615	0.156	0.138	1.000	1.000	-0.546	0.447
tTime	0.591	0.470	-0.200	-0.184	-0.511	-0.511	1.000	-0.896
# nodes	-0.676	-0.561	0.000	-0.024	0.460	0.460	-0.506	1.000

Table 3.4: Correlation values for CST_SCT3 using Fibonacci words.

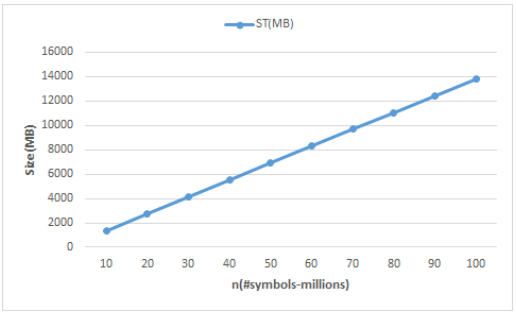
ST								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	0.895	-0.195	-0.239	-0.302	-0.720	-0.922	-0.371
H_1	0.809	1.000	0.030	-0.089	-0.364	-0.625	-0.910	-0.420
maxLCP	-0.218	0.036	1.000	0.959	0.164	0.293	0.017	0.177
meanLCP	-0.161	0.037	0.966	1.000	0.265	0.277	0.099	0.276
size	-0.417	-0.566	0.149	0.154	1.000	0.444	0.320	0.997
cTime	-0.684	-0.606	0.116	0.120	0.416	1.000	0.603	0.493
tTime	-0.731	-0.629	0.068	0.047	0.356	0.453	1.000	0.379
# nodes	-0.676	-0.561	0.000	-0.024	0.463	0.699	0.353	1.000

Table 3.5: Correlation values for ST using Fibonacci words.

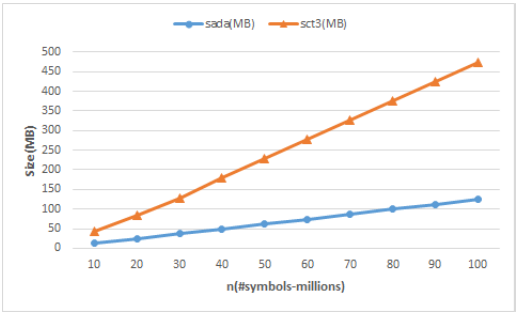
Using the correlation values from Tables 3.3, 3.4, and 3.5 for Fibonacci words, it can be observed that, the size of the suffix data structures vary inversely w.r.t varying entropy values, more significantly in the case of CSTs and less significantly in the case of ST. The construction time varies inversely w.r.t to varying entropy values for all the suffix data structures. The traversal time has a positive correlation w.r.t entropy values in the case of compressed suffix trees and high negative correlation in the case of suffix trees. The LCP values have a negative correlation w.r.t the traversal time in the case of CSTs. The size and construction time for the data structures has a positive correlation w.r.t the number of the nodes in the structure, whereas the traversal time has a negative correlation in the case of CSTs and positive correlation in the case of ST. In particular, for ST the size has an almost perfect correlation (ρ=0.997) with the number of nodes(# nodes).

Figure 3.3 and Table 3.6 show that, in general, with increasing data

sizes, the size of the suffix data structures are linear with respect to n, the data size. Table 3.6 illustrates that, on average, the space required by suffix tree (ST) is approximately 111.21 times that for CST_SADA and approximately 30.61 times that for CST_SCT3. The per symbol values remain fairly constant in the case of suffix tree with increasing data sizes, but increases in the case of CST_SCT3, and in the case of CST_SADA it does not show any reasonable behavior.

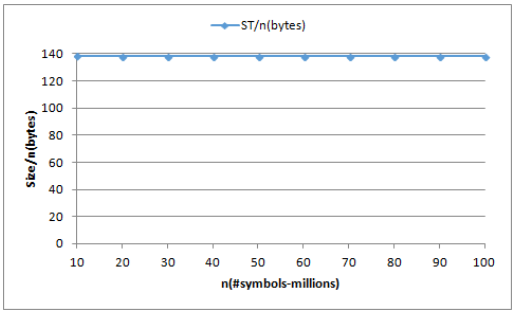


(a) ST

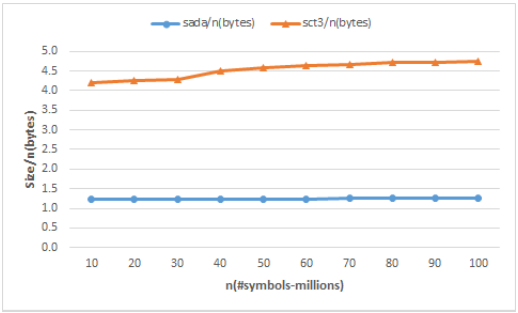


(b) CST

Figure 3.3: Size(MB) of suffix data structures with varying data sizes(n) using Fibonacci words.



(a) ST

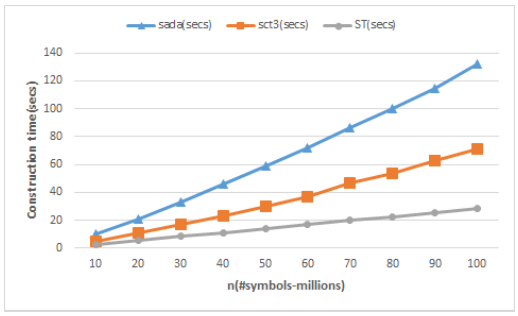


(b) CST

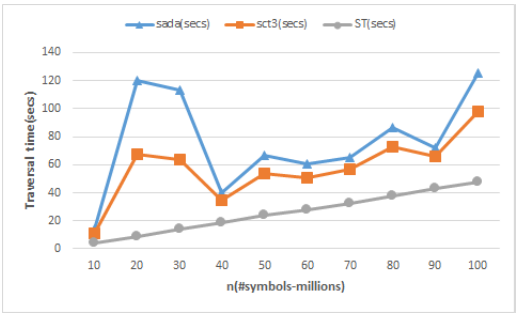
Figure 3.4: Size per symbol(bytes) of suffix data structures with varying data sizes(n) using Fibonacci words.

Figure 3.5a and Table 3.7 show that, with increasing data sizes the time required to construct the data structures are generally linear with

respect to n, the data size. Table 3.7 illustrates that, on average, the construction time required for CST_SADA is around 4.18 times that for the suffix tree (ST), and about 1.93 times that for CST_SCT3. The per symbol values increase in the case of CST_SADA and CST_SCT3 with the linear increase in data sizes. However, in the case of suffix tree it is fairly constant with slight variation which does not show any reasonable behavior.

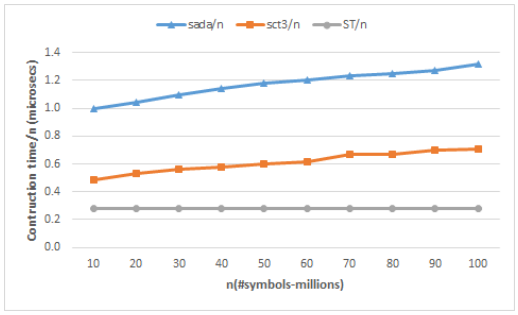


(a) Construction time

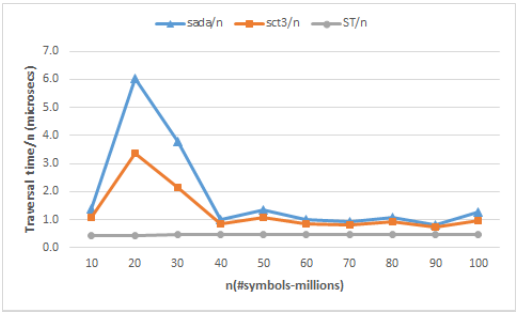


(b) Traversal time

Figure 3.5: Time requirements(secs) for suffix data structures with varying data sizes(n) using Fibonacci words.



(a) Construction time



(b) Traversal time

Figure 3.6: Time requirements per symbol(μsecs) for suffix data structures with varying data sizes(n) using Fibonacci words.

Figure 3.5b and Table 3.8 show the time required to traverse the respective structures to generate the occurrence vectors. While the traversal time varies linearly with respect to n for the suffix tree, it is not exactly

linear for CST_SADA and CST_SCT3. This behavior is observed due to the time complexity involved in calculating the suffix number for a leaf node with the use of operation $sn(v)$ defined in the sdsl library. Table 3.8 illustrates that, on average, the traversal time required for CST_SADA is around 4.07 times that for the suffix tree (ST), and about 1.31 times the traversal time required for CST_SCT3. The per symbol values increase in the case of suffix tree, and decrease in the case of compressed suffix trees. Higher average LCP leads to shorter traversal time for CST, but not for ST. Higher LCP values often imply more compressibility.

n(#symbols)	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
10,000	0.01	0.03	1.38	1.216	2.934	138.286	113.73	47.13	2.41
100,000	0.14	0.33	13.83	1.357	3.305	138.282	101.91	41.84	2.44
1,000,000	1.23	3.65	138.28	1.225	3.648	138.283	112.85	37.90	2.98
10,000,000	12.32	41.98	1382.97	1.232	4.197	138.283	112.30	32.95	3.41
20,000,000	24.75	85.10	2765.65	1.238	4.255	138.282	111.73	32.50	3.44
30,000,000	37.02	128.13	4148.48	1.234	4.271	138.283	112.06	32.38	3.46
40,000,000	49.75	179.78	5531.31	1.244	4.495	138.283	111.19	30.77	3.61
50,000,000	62.14	228.47	6914.14	1.243	4.569	138.283	111.27	30.26	3.68
60,000,000	74.50	277.59	8296.97	1.242	4.627	138.283	111.36	29.89	3.73
70,000,000	87.64	327.24	9679.79	1.252	4.675	138.283	110.45	29.58	3.73
80,000,000	100.07	376.43	11062.62	1.251	4.705	138.283	110.55	29.39	3.76
90,000,000	112.53	425.61	12445.45	1.250	4.729	138.283	110.60	29.24	3.78
100,000,000	124.99	474.79	13828.28	1.250	4.748	138.283	110.64	29.12	3.80
			Average	1.24	4.53	138.28	111.21	30.61	3.64
			Ave_Dev	0.006	0.178	0.002	0.529	1.232	0.128

Table 3.6: Size(MB) of suffix data structures with varying data sizes(n) using Fibonacci words.

n(#symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
10,000	0.030	0.024	0.002	3.019	2.392	0.249	12.14	9.62	1.26
100,000	0.088	0.044	0.026	0.880	0.437	0.260	3.39	1.68	2.01
1,000,000	0.753	0.305	0.278	0.753	0.305	0.278	2.71	1.10	2.47
10,000,000	9.95	4.85	2.78	0.995	0.485	0.278	3.58	1.74	2.05
20,000,000	20.91	10.56	5.56	1.046	0.528	0.278	3.76	1.90	1.98
30,000,000	32.89	16.82	8.37	1.096	0.561	0.279	3.93	2.01	1.96
40,000,000	45.74	23.21	11.18	1.143	0.580	0.280	4.09	2.08	1.97
50,000,000	58.94	30.16	13.96	1.179	0.603	0.279	4.22	2.16	1.95
60,000,000	71.94	37.08	16.96	1.199	0.618	0.283	4.24	2.19	1.94
70,000,000	86.49	46.61	19.78	1.236	0.666	0.283	4.37	2.36	1.86
80,000,000	99.74	53.39	22.60	1.247	0.667	0.282	4.41	2.36	1.87
90,000,000	114.48	63.11	25.33	1.272	0.701	0.281	4.52	2.49	1.81
100,000,000	131.97	70.87	28.10	1.320	0.709	0.281	4.70	2.52	1.86
			Average	1.17	0.61	0.28	4.18	2.18	1.93
			Ave_Dev	0.082	0.060	0.002	0.274	0.203	0.060

Table 3.7: Variation of construction time(secs) for suffix data structures with varying data sizes(n) using Fibonacci words.

n(#symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
10,000	0.014	0.010	0.001	1.426	1.031	0.135	10.54	7.63	0.72
100,000	0.142	0.102	0.023	1.421	1.017	0.234	6.06	4.34	0.72
1,000,000	2.300	1.494	0.296	2.300	1.494	0.296	7.78	5.05	0.65
10,000,000	13.89	10.93	4.23	1.389	1.093	0.423	3.28	2.58	0.79
20,000,000	120.40	67.36	8.95	6.020	3.368	0.447	13.46	7.53	0.56
30,000,000	113.07	64.04	13.75	3.769	2.135	0.458	8.23	4.66	0.57
40,000,000	40.35	34.58	18.48	1.009	0.865	0.462	2.18	1.87	0.86
50,000,000	66.68	54.11	23.64	1.334	1.082	0.473	2.82	2.29	0.81
60,000,000	60.33	50.76	27.88	1.005	0.846	0.465	2.16	1.82	0.84
70,000,000	64.91	56.54	32.54	0.927	0.808	0.465	1.99	1.74	0.87
80,000,000	86.55	72.75	37.96	1.082	0.909	0.475	2.28	1.92	0.84
90,000,000	71.71	66.23	43.14	0.797	0.736	0.479	1.66	1.54	0.92
100,000,000	125.20	97.65	47.87	1.252	0.976	0.479	2.62	2.04	0.78
			Average	1.86	1.28	0.46	4.07	2.80	0.78
			Ave_Dev	1.214	0.588	0.012	2.710	1.318	0.089

Table 3.8: Variation of traversal time(secs) for suffix data structures with varying data sizes(n) using Fibonacci words.

3.4.2 Real Data

Table 3.9 shows the LCP and entropy information for the real data set obtained from the *Pizza & Chili* corpus [18]. The files marked with $*$ are real data sets obtained by taking only the first 100 MB of each file. As observed, the meanLCP is highest for the file $kernel^*$ and lowest for the file $influenza^*$.

FileName	$\|\Sigma\|$	n(#symbols)	#numNodes	#numNodes/n	maxLCP	meanLCP	maxLCP/n	meanLCP/n	H_0	H_1
Escherichia_Coli	15	112,689,515	217,430,271	1.929	698,433	11,322	0.006198	0.000100	2.00	1.98
world_leaders	89	46,968,181	89,618,348	1.908	695,051	8,837	0.014798	0.000188	3.47	1.95
einstein.de.txt	117	92,758,441	183,662,236	1.980	258,006	35,248	0.002781	0.000380	5.04	3.59
$cere^*$	5	104,857,600	202,359,830	1.930	32,469	2,130	0.000310	0.000020	2.20	1.80
$para^*$	5	104,857,600	201,676,800	1.923	42,013	1,300	0.000401	0.000012	2.13	1.87
$Escherichia\_Coli^*$	15	104,857,600	201,964,284	1.926	698,433	11,870	0.006661	0.000113	2.00	1.98
$influenza^*$	15	104,857,600	204,446,109	1.950	23,483	815	0.000224	0.000008	1.97	1.93
$einstein.en^*$	139	104,857,600	205,375,456	1.959	1,738,784	49,722	0.016582	0.000474	4.86	3.64
$kernel^*$	160	104,857,600	205,909,946	1.964	1,413,731	108,325	0.013482	0.001033	5.35	4.09
$coreutils^*$	236	104,857,600	193,138,002	1.842	2,172,045	82,654	0.020714	0.000788	5.43	4.12

Table 3.9: LCP, entropy and other attributes for real data set obtained from the *Pizza & Chili* corpus [18]

Note: For the file Escherichia_Coli the alphabet size is 15. Although there are four bases {A,C,G,T}, DNA sequences may have alphabets of size up to $16 = 2^4$ because some characters denote an unknown choice among the four bases. The most common character used is N, which denotes a totally unknown symbol [18].

CST_SADA									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.966	0.972	0.954	0.908	-0.688	-0.942	-0.547	-0.418
H_0	0.551	1.000	0.995	0.928	0.946	-0.682	-0.965	-0.601	-0.200
H_1	0.951	0.524	1.000	0.938	0.959	-0.689	-0.948	-0.601	-0.205
maxLCP	0.651	0.619	0.714	1.000	0.847	-0.612	-0.862	-0.492	-0.365
meanLCP	0.651	0.714	0.619	0.714	1.000	-0.552	-0.932	-0.473	-0.146
size	-0.451	-0.429	-0.333	-0.238	-0.143	1.000	0.564	0.963	-0.004
cTime	-0.551	-1.000	-0.524	-0.619	-0.714	0.429	1.000	0.460	0.314
tTime	-0.250	-0.238	-0.143	-0.048	-0.143	0.810	0.238	1.000	-0.264
# nodes	0.250	0.048	0.143	-0.143	0.143	-0.238	-0.048	-0.429	1.000

Table 3.10: Correlation values for CST_SADA using real data.

CST_SCT3									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.966	0.972	0.954	0.908	0.858	-0.892	0.850	-0.418
H_0	0.551	1.000	0.995	0.928	0.946	0.956	-0.911	0.824	-0.200
H_1	0.951	0.524	1.000	0.938	0.959	0.944	-0.901	0.830	-0.205
maxLCP	0.651	0.619	0.714	1.000	0.847	0.853	-0.748	0.780	-0.365
meanLCP	0.651	0.714	0.619	0.714	1.000	0.892	-0.905	0.891	-0.146
size	0.551	0.524	0.429	0.333	0.619	1.000	-0.845	0.666	0.081
cTime	-0.551	-0.714	-0.429	-0.333	-0.619	-0.810	1.000	-0.791	0.217
tTime	0.651	0.810	0.714	0.810	0.714	0.333	-0.524	1.000	-0.491
# nodes	0.250	0.048	0.143	-0.143	0.143	0.524	-0.333	-0.143	1.000

Table 3.11: Correlation values for CST_SCT3 using real data.

ST									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.966	0.972	0.954	0.908	-0.421	-0.376	-0.772	-0.418
H_0	0.551	1.000	0.995	0.928	0.946	-0.204	-0.450	-0.688	-0.200
H_1	0.951	0.524	1.000	0.938	0.959	-0.209	-0.457	-0.688	-0.205
maxLCP	0.651	0.619	0.714	1.000	0.847	-0.368	-0.368	-0.705	-0.365
meanLCP	0.651	0.714	0.619	0.714	1.000	-0.150	-0.348	-0.497	-0.146
size	0.250	0.048	0.143	-0.143	0.143	1.000	-0.431	0.397	1.000
cTime	-0.350	-0.143	-0.238	-0.143	-0.238	-0.524	1.000	0.599	-0.436
tTime	-0.551	-0.429	-0.524	-0.429	-0.143	-0.048	0.524	1.000	0.393
# nodes	0.250	0.048	0.143	-0.143	0.143	1.000	-0.524	-0.048	1.000

Table 3.12: Correlation values for ST using real data.

The correlation tables are obtained using only the files marked with a '*'. Using the correlation values from Tables 3.10, 3.11, and 3.12 for real data, it can be observed that $|\Sigma|$, H_0, LCP values and number of nodes have negative correlation w.r.t the size of CST_SADA and ST and positive correlation w.r.t the size of CST_SCT3. The construction time of CSTs and

ST vary inversely w.r.t the entropy H_0, alphabet size $|\Sigma|$ and the LCP values. The traversal time of CST_SADA and ST has a negative correlation w.r.t the H_0, $|\Sigma|$ and LCP values and positive correlation for CST_SCT3. For this data set, the size for ST is perfectly correlated with the number of nodes(ρ=1, τ=1).

Table 3.13 illustrates that, the space required by ST is at least 99 times the space required by CST_SADA and at least 35 times the space required by CST_SCT3. Also, the space required by CST_SCT3 is at least 1.85 times the space required by CST_SADA.

FileName	$\|\Sigma\|$	n(symbols)	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
Escherichia_Coli	15	112,689,515	150.24	283.55	15037.22	1.333	2.516	133.439	100.09	53.03	1.89
world_leaders	89	46,968,181	57.71	141.80	6198.40	1.229	3.019	131.970	107.41	43.71	2.46
einstein.de.txt	117	92,758,441	115.53	360.95	12699.55	1.245	3.891	136.910	109.93	35.18	3.12
*cere**	5	104,857,600	137.84	276.76	13994.95	1.315	2.639	133.466	101.53	50.57	2.01
*para**	5	104,857,600	139.86	258.24	13948.05	1.334	2.463	133.019	99.73	54.01	1.85
*Escherichia_Coli**	15	104,857,600	140.18	261.98	13967.79	1.337	2.498	133.207	99.64	53.32	1.87
*influenza**	15	104,857,600	131.10	267.62	14138.20	1.250	2.552	134.832	107.84	52.83	2.04
*einstein.en**	139	104,857,600	129.81	409.34	14202.01	1.238	3.904	135.441	109.41	34.70	3.15
*kernel**	160	104,857,600	132.97	412.03	14238.71	1.268	3.929	135.791	107.08	34.56	3.10
*coreutils**	236	104,857,600	130.98	364.78	13361.73	1.249	3.479	127.427	102.01	36.63	2.78
					Average	1.28	3.09	133.55	104.47	44.85	2.43
					Ave_Dev	0.040	0.569	1.755	3.867	7.898	0.497

Table 3.13: Size(MB) of suffix data structures using real data set obtained from the *Pizza & Chili* corpus [18]

Table 3.14 illustrates that, the construction time required for CST_SADA is at least 1.8 times the construction time required for ST. The construction time required for CST_SCT3 is approximately equal to the time required for ST for certain files and is less than 2 times for the other files. Also, the construction time required for CST_SADA is at least twice the construction time required for CST_SCT3.

FileName	$\|\Sigma\|$	n(symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
Escherichia_Coli	15	112,689,515	123.08	59.83	59.70	1.092	0.531	0.530	2.06	1.00	2.06
world_leaders	89	46,968,181	43.41	17.24	15.63	0.924	0.367	0.333	2.78	1.10	2.52
einstein.de.txt	117	92,758,441	89.33	42.61	25.84	0.963	0.459	0.279	3.46	1.65	2.10
*cere**	5	104,857,600	107.62	49.78	49.53	1.026	0.475	0.472	2.17	1.01	2.16
*para**	5	104,857,600	109.60	52.19	60.94	1.045	0.498	0.581	1.80	0.86	2.10
*Escherichia_Coli**	15	104,857,600	111.60	53.39	54.22	1.064	0.509	0.517	2.06	0.98	2.09
*influenza**	15	104,857,600	112.14	50.44	32.67	1.069	0.481	0.312	3.43	1.54	2.22
*einstein.en**	139	104,857,600	102.92	48.21	28.54	0.982	0.460	0.272	3.61	1.69	2.13
*kernel**	160	104,857,600	98.82	44.61	38.70	0.942	0.425	0.369	2.55	1.15	2.22
*coreutils**	236	104,857,600	97.77	44.86	46.90	0.932	0.428	0.447	2.08	0.96	2.18
					Average	1.00	0.46	0.41	2.60	1.19	2.18
					Ave_Dev	0.055	0.035	0.098	0.575	0.260	0.085

Table 3.14: Construction time(secs) for suffix data structures using real data set obtained from the *Pizza & Chili* corpus [18]

Table 3.15 illustrates that, the traversal time required for CST_SADA is at least 3.6 times the traversal time required for ST, however, the traversal time for certain files *Escherichia_Coli*, *cere**, *para**, and *Escherichia_Coli** is more than expected in the case of CST_SADA. This behavior is similar to the case of Fibonacci words with small alphabet size. The traversal time required for CST_SCT3 is at least 4.04 times the traversal time required for ST. The time required for CST_SADA is more in comparison to CST_SCT3 for the files with small alphabet size which is similar to the case of Fibonacci words, and the opposite for the remaining files.

FileName	$\|\Sigma\|$	n(symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
Escherichia_Coli	15	112,689,515	620.12	346.95	47.92	5.503	3.079	0.425	12.94	7.24	0.56
world_leaders	89	46,968,181	94.91	119.25	18.27	2.021	0.367	0.389	5.20	6.53	1.26
einstein.de.txt	117	92,758,441	134.96	266.21	37.07	1.455	2.870	0.400	3.64	7.18	1.97
*cere**	5	104,857,600	445.72	266.70	42.12	4.251	2.543	0.402	10.58	6.33	0.60
*para**	5	104,857,600	530.09	290.78	42.00	5.055	2.773	0.401	12.62	6.92	0.55
*Escherichia_Coli**	15	104,857,600	546.00	288.11	42.77	5.207	2.748	0.408	12.77	6.74	0.53
*influenza**	15	104,857,600	222.81	162.63	40.25	2.125	1.551	0.384	5.54	4.04	0.73
*einstein.en**	139	104,857,600	150.71	294.87	39.28	1.437	2.812	0.375	3.84	7.51	1.96
*kernel**	160	104,857,600	275.74	489.84	41.27	2.630	4.671	0.394	6.68	11.87	1.78
*coreutils**	236	104,857,600	318.48	525.87	38.41	3.037	5.015	0.366	8.29	13.69	1.65
					Average	3.27	2.84	0.39	8.21	7.80	1.16
					Ave_Dev	1.386	0.853	0.013	3.231	1.990	0.565

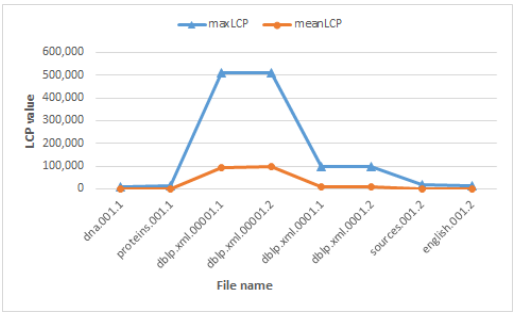
Table 3.15: Traversal time(secs) for suffix data structures using real data set obtained from the *Pizza & Chili* Corpus [18]

3.4.3 Pseudo Real Data

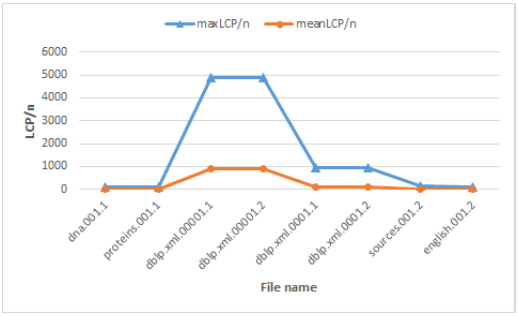
Table 3.16 and Figure 3.7 show the LCP and entropy information for the pseudo real data set obtained from the *Pizza & Chili* corpus [18] for different files, each of size n=100MB (104,857,600 symbols). The meanLCP is high for XML data, and small for DNA, protein, english text files, and source files (C/Java source code).

FileName	$\|\Sigma\|$	#numNodes	#numNodes/n	maxLCP	meanLCP	maxLCP/n ($X10^{-6}$)	meanLCP/n ($X10^{-6}$)	H_0	H_1
dna.001.1	5	207,298,926	1.977	10,899	992.995	103.94	9.470	1.997	1.942
proteins.001.1	21	204,885,653	1.954	11,091	991.396	105.77	9.455	4.184	4.169
dblp.xml.00001.1	89	209,151,185	1.995	510,561	94981.5	4869.09	905.814	5.215	3.087
dblp.xml.00001.2	89	209,151,286	1.995	510,561	95781.4	4869.09	913.443	5.215	3.094
dblp.xml.0001.1	89	208,772,302	1.991	97,633	9844.75	931.10	93.887	5.215	3.089
dblp.xml.0001.2	89	208,986,630	1.993	97,633	9865.58	931.10	94.086	5.215	3.145
sources.001.2	98	208,329,606	1.987	16,529	992.218	157.63	9.463	5.500	4.099
english.001.2	106	208,331,277	1.987	11,222	987.289	107.02	9.416	4.587	3.820

Table 3.16: *LCP*, entropy and other attributes for pseudo real data obtained from the *Pizza & Chili* corpus [18].



(a) meanLCP and maxLCP



(b) meanLCP/n and maxLCP/n

Figure 3.7: LCP information for pseudo real data set obtained from the *Pizza & Chili* corpus [18].

CST_SADA									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.852	0.373	0.318	0.288	-0.093	-0.764	-0.378	0.768
H_0	0.545	1.000	0.584	0.382	0.346	-0.412	-0.636	-0.600	0.506
H_1	0.322	0.081	1.000	-0.217	-0.204	0.174	-0.237	0.152	-0.291
maxLCP	0.167	0.585	-0.297	1.000	0.997	-0.637	-0.188	-0.709	0.526
meanLCP	-0.161	0.242	-0.357	0.667	1.000	-0.590	-0.169	-0.655	0.489
size	0.171	-0.256	0.340	-0.628	-0.718	1.000	-0.055	0.945	-0.239
cTime	-0.564	-0.645	-0.286	-0.297	-0.214	-0.113	1.000	0.232	-0.681
tTime	0.081	-0.322	0.214	-0.741	-0.714	0.869	0.071	1.000	-0.525
# nodes	0.161	0.403	-0.357	0.815	0.714	-0.416	-0.357	-0.571	1.000

Table 3.17: Correlation values for CST_SADA using pseudo real data.

CST_SCT3									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.852	0.373	0.318	0.288	0.615	-0.325	0.900	0.768
H_0	0.545	1.000	0.584	0.382	0.346	0.731	-0.647	0.936	0.506
H_1	0.322	0.081	1.000	-0.217	-0.204	-0.022	-0.286	0.692	-0.291
maxLCP	0.167	0.585	-0.296	1.000	0.997	0.856	-0.264	0.155	0.526
meanLCP	-0.161	0.242	-0.357	0.667	1.000	0.818	-0.233	0.131	0.489
size	0.164	0.574	-0.327	0.981	0.618	1.000	-0.506	0.501	0.683
cTime	-0.246	-0.656	-0.036	-0.302	-0.255	-0.333	1.000	-0.482	-0.222
tTime	0.806	0.564	0.429	0.222	-0.071	0.182	-0.109	1.000	0.475
# nodes	0.161	0.403	-0.357	0.815	0.714	0.764	-0.109	0.071	1.000

Table 3.18: Correlation values for CST_SCT3 using pseudo real data.

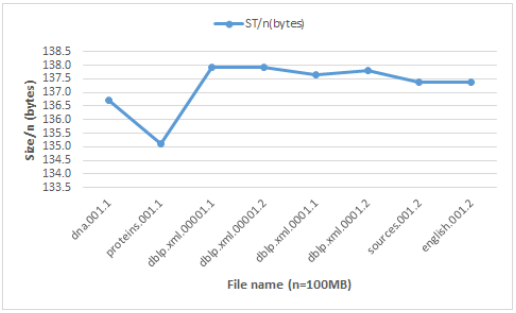
ST									
	$\|\Sigma\|$	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	1.000	0.852	0.373	0.318	0.288	0.768	0.128	0.514	0.768
H_0	0.545	1.000	0.584	0.382	0.346	0.506	-0.066	0.306	0.506
H_1	0.322	0.081	1.000	-0.217	-0.204	-0.291	0.557	0.300	-0.291
maxLCP	0.167	0.585	-0.296	1.000	0.997	0.526	-0.624	0.024	0.526
meanLCP	-0.161	0.242	-0.357	0.667	1.000	0.489	-0.582	0.033	0.489
size	0.161	0.403	-0.357	0.815	0.714	1.000	-0.263	0.323	1.000
cTime	0.246	-0.164	0.473	-0.566	-0.764	-0.473	1.000	0.417	-0.263
tTime	0.403	0.000	0.214	0.074	-0.143	0.143	0.400	1.000	0.323
# nodes	0.161	0.403	-0.357	0.815	0.714	1.000	-0.473	0.143	1.000

Table 3.19: Correlation values for ST using pseudo real data.

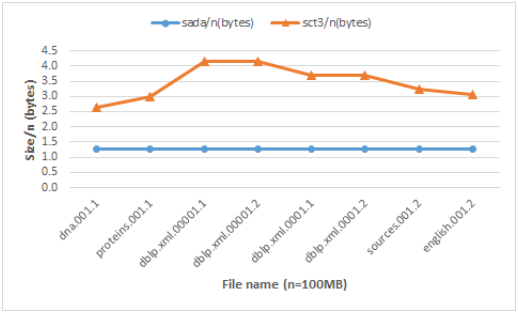
Using the correlation values from Tables 3.17, 3.18, and 3.19 for pseudo real data, it can be observed that $|\Sigma|$, H_0, LCP values and number of nodes have negative correlation w.r.t the size of CST_SADA and positive correlation w.r.t the size of CST_SCT3 and ST. The construction time of CSTs vary inversely w.r.t the entropy H_0 and alphabet size $|\Sigma|$. The construction time of ST vary directly w.r.t the entropy H_1 and vary inversely w.r.t the LCP values. The traversal time of CST_SADA has a negative correlation w.r.t the H_0 and $|\Sigma|$. The traversal time of CST_SCT3 and ST has a high positive correlation w.r.t the H_0 and $|\Sigma|$. Also, the traversal time of CST_SADA has a negative correlation w.r.t the number of nodes. For this data set, the size for ST is perfectly correlated with the number of nodes(ρ=1, τ=1).

Table 3.16 explains Figure 3.8a depicting the size of the suffix tree structure for different files which is proportional to the number of nodes in the data structure built for the specific data file. Figure 3.8b shows that the size of the CST_SADA does not vary with the nature of data, however,

it varies in the case of CST_SCT3. Table 3.20 illustrates that, on average, the space required by suffix tree (ST) is approximately 109.32 times that for CST_SADA and 40.68 times that for CST_SCT3.



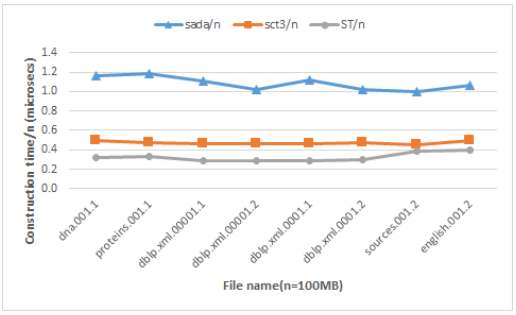
(a) ST



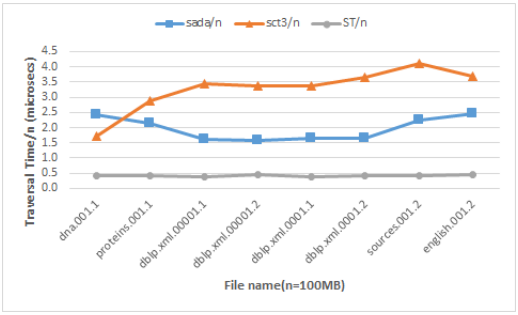
(b) CST

Figure 3.8: Size(MB) of suffix data structures using pseudo real data obtained from the *Pizza & Chili* corpus [18].

Figure 3.9a and Table 3.21 show that, in general, the time required to construct the data structures does not vary much for different files of similar data size. Table 3.21 illustrates that, on average, the construction time required for CST_SADA is around 3.39 times that for the suffix tree (ST), and about 2.29 times that for CST_SCT3.



(a) Construction time



(b) Traversal time

Figure 3.9: Time requirements(secs) for suffix data structures using pseudo real data obtained from the *Pizza & Chili* corpus [18].

Figure 3.9b and Table 3.22 show that, for ST the time required to

traverse the data structure does not vary much for different files of same data size. For CST_SCT3 it increased with increasing entropy, and for CST_SADA it decreased with increasing entropy upto a point. Table 3.22 illustrates that on average the traversal time required for CST_SADA is around 4.75 times that for the suffix tree (ST), and about 0.575 times that for CST_SCT3. The traversal time required for CST_SADA is more when compared to CST_SCT3 only in the case of DNA sequences (small alphabet size).

FileName	$\|\Sigma\|$	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
dna.001.1	5	132.23	276.74	14334.09	1.261	2.639	136.701	108.41	51.80	2.09
proteins.001.1	21	131.49	313.99	14168.38	1.254	2.994	135.120	107.75	45.12	2.39
dblp.xml.00001.1	89	131.08	434.25	14461.27	1.250	4.141	137.913	110.32	33.30	3.31
dblp.xml.00001.2	89	131.10	434.26	14461.28	1.250	4.141	137.914	110.31	33.30	3.31
dblp.xml.0001.1	89	131.11	388.22	14435.26	1.250	3.702	137.665	110.10	37.18	2.96
dblp.xml.0001.2	89	131.22	387.99	14449.97	1.251	3.700	137.806	110.12	37.24	2.96
sources.001.2	98	132.28	337.99	14404.86	1.262	3.223	137.375	108.90	42.62	2.56
english.001.2	106	132.60	320.79	14404.97	1.265	3.059	137.377	108.63	44.91	2.42
				Average	1.26	3.45	137.23	109.32	40.68	2.75
				Ave_Dev	0.01	0.47	0.66	0.90	5.43	0.39

Table 3.20: Size(MB) of suffix data structures using pseudo real data set obtained from the *Pizza & Chili* corpus [18].

FileName	$\|\Sigma\|$	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
dna.001.1	5	121.38	51.60	33.73	1.158	0.492	0.322	3.60	1.53	2.35
proteins.001.1	21	124.03	49.67	34.96	1.183	0.474	0.333	3.55	1.42	2.50
dblp.xml.00001.1	89	116.60	49.03	30.02	1.112	0.468	0.286	3.88	1.63	2.38
dblp.xml.00001.2	89	107.48	49.02	30.22	1.025	0.468	0.288	3.56	1.62	2.19
dblp.xml.0001.1	89	117.19	47.92	30.18	1.118	0.457	0.288	3.88	1.59	2.45
dblp.xml.0001.2	89	106.68	49.92	31.41	1.017	0.476	0.300	3.40	1.59	2.14
sources.001.2	98	104.70	47.44	40.75	0.998	0.452	0.389	2.57	1.16	2.21
english.001.2	106	111.97	52.28	41.54	1.068	0.499	0.396	2.70	1.26	2.14
				Average	1.08	0.47	0.33	3.39	1.48	2.29
				Ave_Dev	0.06	0.01	0.04	0.38	0.15	0.12

Table 3.21: Construction time(secs) for suffix data structures using pseudo real data set obtained from the *Pizza & Chili* corpus [18].

FileName	$\|\Sigma\|$	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
dna.001.1	5	253.04	180.88	41.93	2.413	1.725	0.400	6.04	4.31	0.71
proteins.001.1	21	225.44	303.25	42.17	2.150	2.892	0.402	5.35	7.19	1.35
dblp.xml.00001.1	89	170.04	359.12	41.53	1.622	3.425	0.396	4.09	8.65	2.11
dblp.xml.00001.2	89	167.05	354.23	45.81	1.593	3.378	0.437	3.65	7.73	2.12
dblp.xml.0001.1	89	171.51	353.75	41.23	1.636	3.374	0.393	4.16	8.58	2.06
dblp.xml.0001.2	89	173.44	383.19	45.23	1.654	3.654	0.431	3.83	8.47	2.21
sources.001.2	98	233.85	429.41	44.20	2.230	4.095	0.422	5.29	9.72	1.84
english.001.2	106	259.30	384.72	46.29	2.473	3.669	0.441	5.60	8.31	1.48
				Average	1.97	3.28	0.42	4.75	7.87	1.74
				Ave_Dev	0.35	0.48	0.02	0.82	1.09	0.42

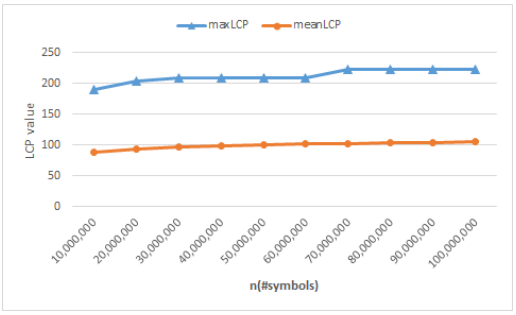
Table 3.22: Traversal time(secs) for suffix data structures using pseudo real data set obtained from the *Pizza & Chili* corpus [18].

3.4.4 Synthetic Data

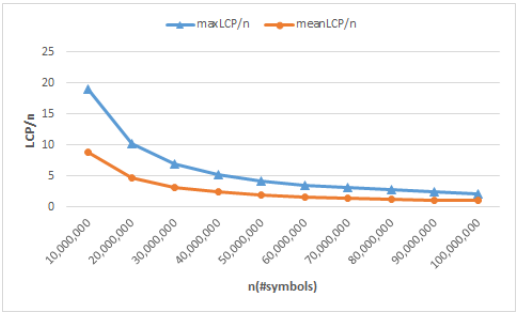
Table 3.23 shows the LCP values and entropy information for synthetic data. Since the data is generated with a small period value of 32, the LCP values are relatively low as compared to those of the Fibonacci words. However, with an increase in data size n, there is a linear increase in the meanLCP values. The meanLCP per symbol decreases with an increase in data size, and this can contribute to slow growth of the meanLCP with increasing data sizes (n). The values of maxLCP and maxLCP per symbol follow similar trends. The entropy is generally constant, independent of the data size n.

n(#symbols)	#numNodes	#numNodes/n	maxLCP	meanLCP	maxLCP/n (X 10^{-6})	meanLCP/n 10^{-6})	H_0	H_1
10,000,000	14182915	1.418	191	88.522	19.100	8.852	3.04627	1.92298
20,000,000	28515660	1.426	204	93.503	10.200	4.675	3.04627	1.92298
30,000,000	43187797	1.440	210	96.528	7.000	3.218	3.04624	1.92305
40,000,000	57684347	1.442	210	98.702	5.250	2.468	3.04623	1.92302
50,000,000	71919378	1.438	210	100.383	4.200	2.008	3.04622	1.92305
60,000,000	85934222	1.432	210	101.731	3.500	1.696	3.04619	1.92307
70,000,000	99846184	1.426	223	102.856	3.186	1.469	3.04620	1.92308
80,000,000	113729820	1.422	223	103.816	2.788	1.298	3.04619	1.92311
90,000,000	127641319	1.418	223	104.655	2.478	1.163	3.04620	1.92311
100,000,000	141624095	1.416	223	105.399	2.230	1.054	3.04620	1.9231

Table 3.23: LCP, entropy and other attributes for synthetic data.



(a) meanLCP and maxLCP



(b) meanLCP/n and maxLCP/n

Figure 3.10: LCP information for varying data sizes(n) using synthetic data.

CST_SADA								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	-0.930	0.867	0.864	-0.875	-0.924	-0.743	0.115
H_1	-0.732	1.000	-0.835	-0.835	0.802	0.912	0.659	-0.261
maxLCP	0.707	-0.828	1.000	1.000	-0.965	-0.978	-0.697	-0.135
meanLCP	0.707	-0.828	1.000	1.000	-0.964	-0.978	-0.696	-0.131
size	-0.561	0.500	-0.598	-0.598	1.000	0.957	0.777	0.214
cTime	-0.691	0.814	-0.989	-0.989	0.628	1.000	0.707	-0.010
tTime	-0.566	0.460	-0.511	-0.511	0.828	0.539	1.000	0.069
# nodes	0.141	-0.368	0.422	0.422	-0.022	-0.405	0.067	1.000

Table 3.24: Correlation values for CST_SADA using synthetic data.

CST_SCT3								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	-0.930	0.867	0.864	0.572	-0.895	-0.660	0.115
H_1	-0.732	1.000	-0.835	-0.835	-0.547	0.894	0.591	-0.261
maxLCP	0.707	-0.828	1.000	1.000	0.193	-0.988	-0.618	-0.135
meanLCP	0.707	-0.828	1.000	1.000	0.185	-0.988	-0.618	-0.131
size	0.425	-0.317	0.424	0.424	1.000	-0.275	-0.147	0.241
cTime	-0.707	0.828	-1.000	-1.000	-0.424	1.000	0.614	-0.002
tTime	-0.519	0.414	-0.467	-0.467	0.047	0.467	1.000	0.020
# nodes	0.141	-0.368	0.422	0.422	0.471	-0.422	0.111	1.000

Table 3.25: Correlation values for CST_SCT3 using synthetic data.

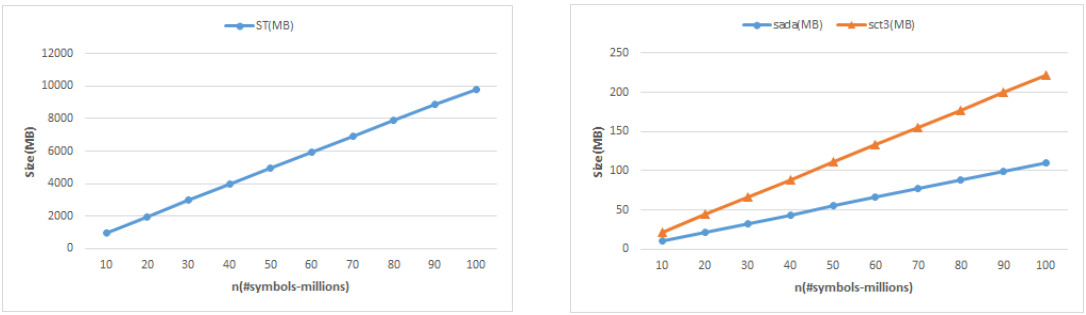
ST								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
H_0	1.000	-0.930	0.867	0.864	0.114	-0.713	-0.920	0.115
H_1	-0.732	1.000	-0.835	-0.835	-0.260	0.618	0.900	-0.261
maxLCP	0.707	-0.828	1.000	1.000	-0.135	-0.900	-0.973	-0.135
meanLCP	0.707	-0.828	1.000	1.000	-0.131	-0.897	-0.972	-0.131
size	0.141	-0.368	0.422	0.422	1.000	0.534	0.019	1.000
cTime	-0.283	0.046	-0.067	-0.067	0.511	1.000	0.854	0.533
tTime	-0.660	0.828	-0.911	-0.911	-0.333	0.156	1.000	0.019
# nodes	0.141	-0.368	0.422	0.422	1.000	0.511	-0.333	1.000

Table 3.26: Correlation values for ST using synthetic data.

Using the correlation values from Tables 3.24, 3.25, and 3.26 for synthetic data, it can be observed that, w.r.t the number of nodes the size of CST_SADA has zero correlation, the size of CST_SCT3 has a positive correlation, and the size of ST has a perfect correlation of 1. Also, the size of

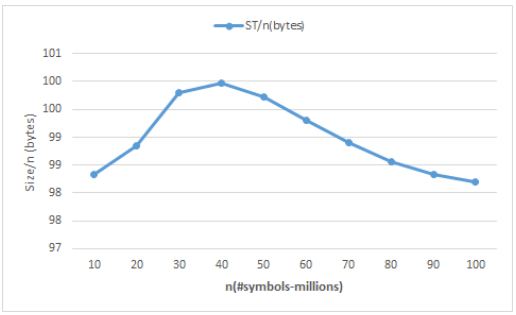
CST_SADA has high positive correlation w.r.t H_1 and high negative correlation w.r.t H_0 and also high negative correlation w.r.t LCP values, whereas, the size of CST_SCT3 has a positive correlation w.r.t H_0 and a negative correlation w.r.t H_1. The construction time of the suffix structures have a high negative correlation w.r.t the LCP values, high negative correlation w.r.t H_0 and high positive correlation w.r.t H_1. The construction time vary inversely w.r.t number of nodes in the case of CST_SADA and CST_SCT3, but vary directly in the case of ST. The traversal time for the suffix data structures have a negative correlation w.r.t H_0, positive correlation w.r.t H_1, negative correlation w.r.t LCP values.

Figure 3.11 and Table 3.27 show that, in general, with increasing data sizes the size of the suffix data structures are linear with respect to n, the data size. Table 3.27 illustrates that, on average, the space required by suffix tree (ST) is approximately 90.15 times that for CST_SADA and approximately 44.58 times that for CST_SCT3.

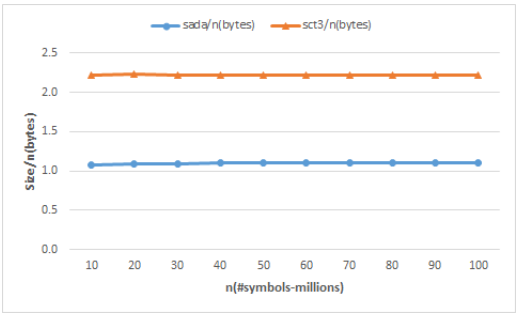


(a) ST (b) CST

Figure 3.11: Size(MB) of suffix data structures using synthetic data.



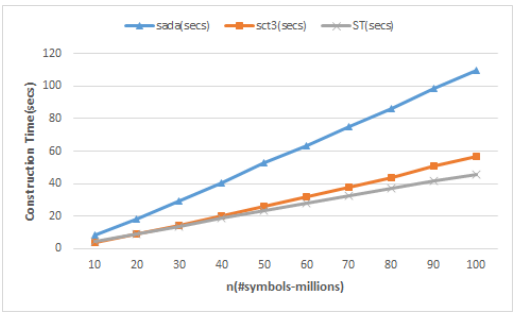
(a) ST



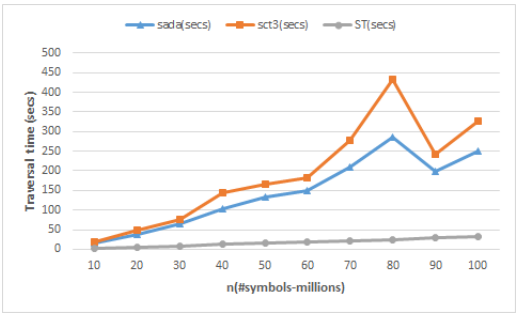
(b) CST

Figure 3.12: Size per symbol(bytes) of suffix data structures using synthetic data.

Figure 3.13a and Table 3.28 show that, in general, with increasing data sizes the time required to construct the data structures are linear with respect to n, the data size. The time per symbol increased linearly with decreasing LCP values per symbol. Table 3.28 illustrates that, on average, the construction time required for CST_SADA is around 2.22 times that for suffix tree (ST) and about 2.01 times that for CST_SCT3.

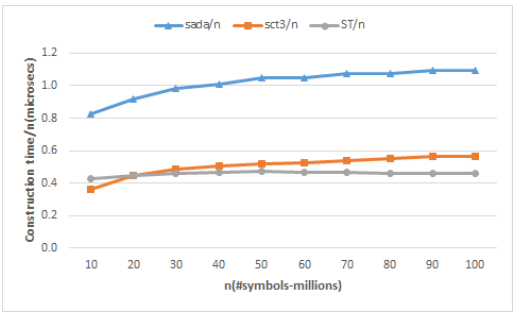


(a) Construction time

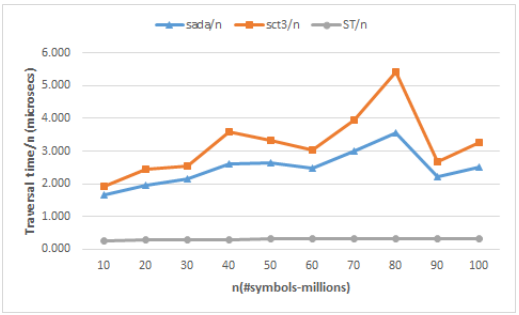


(b) Traversal time

Figure 3.13: Time requirements(secs) for suffix data structures using synthetic data.



(a) Construction time



(b) Traversal time

Figure 3.14: Time requirements per symbol (μsecs) for suffix data structures using synthetic data.

Figure 3.13b and Table 3.29 show the time required for each data structure to traverse the respective structure in order to construct the occurrence vectors. While the traversal time varies linearly with respect to n for the suffix tree, it is not exactly linear for CST_SADA and CST_SCT3. This behavior is observed due to the time complexity involved in calculating the suffix number for the leaf nodes with the use of operation $sn(v)$ defined in the sdsl library. Table 3.29 illustrates that, on average, the traversal time required for CST_SADA is around 8.01 times that for suffix tree (ST) and the traversal time required for CST_SCT3 is around 10.37 times that for suffix tree (ST).

n(#symbols)	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
10,000,000	10.81	22.16	983.40	1.081	2.216	98.340	91.013	44.375	2.051
20,000,000	21.79	44.75	1977.09	1.089	2.237	98.854	90.750	44.185	2.054
30,000,000	32.89	66.69	2994.08	1.096	2.223	99.803	91.037	44.897	2.028
40,000,000	44.06	88.93	3999.02	1.102	2.223	99.975	90.754	44.969	2.018
50,000,000	55.11	110.94	4986.00	1.102	2.219	99.720	90.466	44.944	2.013
60,000,000	66.08	132.95	5957.86	1.101	2.216	99.298	90.159	44.813	2.012
70,000,000	77.32	155.46	6922.65	1.105	2.221	98.895	89.537	44.531	2.011
80,000,000	88.21	177.51	7885.50	1.103	2.219	98.569	89.399	44.422	2.012
90,000,000	99.10	199.56	8850.26	1.101	2.217	98.336	89.303	44.348	2.014
100,000,000	110.28	221.64	9819.92	1.103	2.216	98.199	89.045	44.306	2.010
			Average	1.10	2.22	99.00	90.15	44.58	2.02
			Ave_Dev	0.006	0.004	0.560	0.660	0.262	0.013

Table 3.27: Size(MB) of suffix data structures with varying data sizes (n) using synthetic data.

n(#symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
10,000,000	8.26	3.64	4.26	0.826	0.364	0.426	1.937	0.855	2.266
20,000,000	18.38	9.01	8.90	0.919	0.450	0.445	2.066	1.012	2.041
30,000,000	29.52	14.63	13.86	0.984	0.488	0.462	2.130	1.056	2.018
40,000,000	40.36	20.29	18.75	1.009	0.507	0.469	2.153	1.082	1.989
50,000,000	52.57	26.13	23.56	1.051	0.523	0.471	2.231	1.109	2.012
60,000,000	63.08	31.70	28.04	1.051	0.528	0.467	2.249	1.130	1.990
70,000,000	75.03	37.86	32.51	1.072	0.541	0.464	2.308	1.164	1.982
80,000,000	86.21	43.97	36.89	1.078	0.550	0.461	2.337	1.192	1.960
90,000,000	98.30	50.59	41.64	1.092	0.562	0.463	2.360	1.215	1.943
100,000,000	109.75	56.64	45.92	1.097	0.566	0.459	2.390	1.233	1.938
			Average	1.02	0.51	0.46	2.22	1.10	2.01
			Ave_Dev	0.067	0.044	0.009	0.116	0.083	0.057

Table 3.28: Construction time(secs) of suffix data structures with varying data sizes(n) using synthetic data.

n(#symbols)	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
10,000,000	16.76	19.15	2.64	1.676	1.915	0.264	6.360	7.267	1.143
20,000,000	38.97	48.84	5.71	1.948	2.442	0.285	6.825	8.554	1.253
30,000,000	64.39	76.22	9.00	2.146	2.541	0.300	7.154	8.468	1.184
40,000,000	104.42	143.97	12.24	2.610	3.599	0.306	8.528	11.759	1.379
50,000,000	132.16	166.36	15.86	2.643	3.327	0.317	8.335	10.492	1.259
60,000,000	149.42	181.53	19.06	2.490	3.025	0.318	7.838	9.522	1.215
70,000,000	209.15	276.31	22.40	2.988	3.947	0.320	9.337	12.335	1.321
80,000,000	285.62	433.38	25.49	3.570	5.417	0.319	11.204	17.000	1.517
90,000,000	199.05	241.13	29.57	2.212	2.679	0.329	6.732	8.155	1.211
100,000,000	250.58	326.30	32.20	2.506	3.263	0.322	7.782	10.134	1.302
			Average	2.48	3.22	0.31	8.01	10.37	1.28
			Ave_Dev	0.387	0.695	0.015	1.073	2.022	0.081

Table 3.29: Traversal time(secs)for suffix data structures with varying data sizes(n) using synthetic data.

3.4.5 Summary of Results

Figures 3.15a and 3.15b summarizes the per symbol average and worst case resource requirements respectively, for different data structures using the different test corpora.

- Tables 3.6, 3.13, 3.20, and 3.27 show that the space required by ST is very large as compared to the space required by CST_SADA. On average it is 111.21 times for Fibonacci words, 109.32 times for pseudo real data and 90.15 times for synthetic data and at least 100 times for real data. Also, the space required by ST is considerably large as compared to the space required by CST_SCT3. On average it is 30.61 times for Fibonacci words, 40.68 times for pseudo real data and

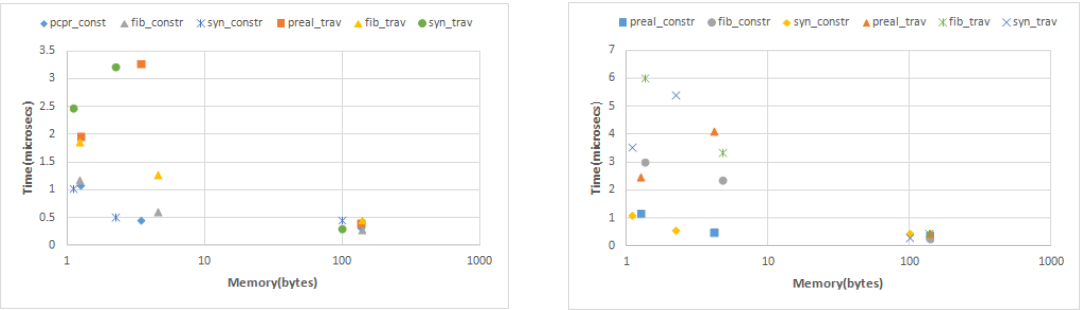
44.58 times for synthetic data and at least 35.18 times for real data. The space required by CST_SCT3 is more when compared to the space required is by CST_SADA. On average, this is 3.64 times for Fibonacci words, 2.75 times for pseudo real data and 2.02 times for synthetic data and at least 1.89 times for real data.

- The construction time required for CSTs is more when compared to the construction time required for ST. The ratio, however, is not in the same order of magnitudes as with the case of the space requirements, where suffix trees require more space compared to CSTs. Also the construction time required for CST_SADA is more when compared to the construction time required for CST_SCT3, which is due to more space required by CST_SCT3 when compared to CST_SADA. Tables 3.7, 3.14, 3.21, and 3.28 show that the construction time required by CST_SADA is more than the construction time required for suffix tree (ST). On average, it is 4.18 times for Fibonacci words, 3.39 times for pseudo real data, 2.22 times for synthetic data, and at least 2.06 times for real data. Tables 3.7, 3.21, and 3.14 show that the construction time for CST_SCT3 on average is 2.18 times to that required for suffix trees using Fibonacci words, 1.48 times using pseudo real data and at most 1.65 times using real data. But Tables 3.14, 3.28 show that they have similar time requirements using real data representing DNA sequences and synthetic data. The construction time required by CST_SADA is more when compared to the construction time required by CST_SCT3. On average it is 1.93 times for Fibonacci words, 2.29 times for pseudo real data and 2.01 times for synthetic data and at least 2.06 times for real data.

- The traversal time required for compressed suffix trees is more when compared to the traversal time required for suffix trees. The ratio, however, is not in the same order of magnitudes as with the case of the space requirements, where suffix trees require much more space compared to compressed suffix trees. Tables 3.8, 3.15, 3.22, and 3.29 show that the traversal time required for CST_SADA is more than the traversal time required for suffix tree (ST). On average, this is 4.07 times for Fibonacci words, 4.75 times for pseudo real data, 8.01 times for synthetic data, and at least 3.64 times for real data. Also, the traversal time required for CST_SCT3 is more than the traversal time

required for suffix tree (ST). On average it is 2.80 times for Fibonacci words, 7.87 times for pseudo real data, 10.37 times for synthetic data, and at least 6.53 times for real data. Also, the traversal time required for CST_SCT3 is more when compared to the traversal time required for CST_SADA, except for Fibonacci words and data sets involving DNA sequences, where the alphabet size is very small. This is opposite to the time requirements for construction of the data structures. On average it is 1.74 times for pseudo real data, 1.28 times for synthetic data and at least 1.26 times for real data. In the case of Fibonacci words, CST_SADA takes on average 1.31 times the time required for CST_SCT3. And in the case of real data representing DNA sequences it is 1.79 times.

- The observations on how the period length influence the performance of the suffix data structures is provided in Appendix B (on Page 88) and the observations on how the alphabet size influence the performance of the suffix data structures is provided in Appendix C (on Page 92). The results are obtained using synthetic data generated in the same manner as generated by Elfeky et al. in [22]



(a) Average case (b) Worst case

Figure 3.15: Summary of resource requirements(per symbol) for suffix data structures using different data sets.

Conclusions

The results show that, compressed suffix trees can replace the suffix trees for space-efficient periodicity mining over large time series data that contain a lot

of repetitions. With the results obtained from the comprehensive empirical analysis, the choice of using a particular variant of compressed suffix trees, namely CST_SADA and CST_SCT3 depends on the application and type of data sets that are used. For general data sets, CST_SCT3 is faster to construct and CST_SADA is faster for generation of occurrence vectors. In the case of Fibonacci words and DNA sequences with small alphabet size, CST_SCT3 is faster in both construction and generation of occurrence vectors.

Figure 3.15 shows the overall resource requirements of the methods on a space-time chart. Best overall result will be the lower-left corner, while positions in the upper-right corner imply the worst performance. As the charts show, there is no clear winner. CSTs using CST_SCT3 seems to provide a middle ground with respect to both time and memory requirements.

Chapter 4

Approximate Periodicity Mining

4.1 A Taxonomy for Approximate Periodicity

Periodicity is a well-studied area and observation of periods in the data leads to valuable discoveries about cyclic events. However, the practical data representing cyclic events rarely exhibit exact periodicity. This leads to the problem of approximate periodicity. The notion of approximate periodicity varies significantly depending on the application, and various definitions have been put forth by different authors. Thus below, we provide a taxonomy for approximate periodicity using two parameters (κ, γ), where κ is the bound on the number of errors allowed in the structure of the periodic pattern, and γ is the bound on the number of errors allowed between the occurrences of a periodic pattern. We begin our discussion with exact periodicity.

4.1.1 Exact periodicity ($\kappa = 0, \gamma = 0$)

For exact periodicity, the same primitive pattern repeats without error using the same exact separation distance between the occurrences each time. We characterize exact periodicity using the patterns $\kappa = 0$ and $\gamma = 0$. Following [7, 8] a periodic string can be defined as follows: Let T be a string of length $n = |T|$. T is called periodic if $T = P^i pref(P)$, where $i \in \mathbb{N}$ and $i \geq 2$, P is a substring of T such that $|P| \leq n/2$, P^i is the concatenation of P to itself i times and $pref(P)$ is a prefix of P. The smallest such substring P is called the period of T. According to [22, 57] the string T is said to have segment periodicity or full periodicity for a pattern P with perfect periodicity.

For example, consider a text $T = abcdefghij\ abcdefghij\ abcdefghij\ abcdefghij\ abcdefghij$. Here, T is said to have perfect segment periodicity for the pattern/substring $P = abcdefghij$ with period length $p = |P| = 10$. The pattern P is periodic with occurrence positions given by its occurrence vector $V = [0, 10, 20, 30, 40]$. Here $\kappa = 0$ (the primitive pattern $P = abcdefghij$ reoccurs in an exact manner with no error), and $\gamma = 0$ (the reoccurring pattern is always separated by the same space, in this case, empty space).

However, practical time series data rarely exhibit segment periodicity. For example, if the time series data represents the daily activities of a person and he daily exercises from 6:00-7:00 AM, gets ready for work from 7:00-8:00 AM and eats his breakfast from 8:00-8:30 AM, drives to work 8:30-9:00 AM, has a meeting from 9:00-10:00 AM and his activities for the rest of the day are unusual. This kind of periodicity is called partial periodicity [22, 57]. The time series is only periodic from 6:00-10:00 AM each day. Let T be a string with length $n = |T|$. T is called partial periodic if $T = (PY)^i pref(P)$, where $i \in \mathbb{N}$ and $i \geq 2$, P and Y are substrings of T such that $|PY| \leq n/2$, P is fixed and Y is any string of fixed length q such that $Y \in \Sigma^q$, where Σ^q represents all the strings of length q over the alphabet Σ, and PY is the concatenation of P to Y. The smallest such substring P is called the partial period of T.

For example, consider a text T=***abcde***$gdiay$ ***abcde***$yiqas$ ***abcde***$opuiu$ ***abcde***$hqeoe$ ***abcde***$lscdf$. Here, T is said to have perfect partial periodic-

ity for the pattern/substring $P = abcde$ with $p = |PY| = 10 \neq (m = |P| = 5)$ and Y is any string of fixed length 5 ($|Y| = p - |P|$). The pattern P is periodic with occurrence positions given by its occurrence vector $V = [0, 10, 20, 30, 40]$. Again $\kappa = 0$ (the primitive pattern $P = abcde$ reoccurs in an exact manner with no error), and $\gamma = 0$ (the reoccurring pattern is always separated by the same space, in this case, 5 each time).

If the string/text T is not periodic, then, it is said to be aperiodic. A string may lose its perfect periodicity due to the errors introduced between the copies of the pattern and/or in the structure of the pattern. Below, we classify the different types of approximate periodicity using the two parameters (κ, γ).

4.1.2 The case of $(\kappa \neq 0, \gamma = 0)$

The approximation is in terms of the structure of the periodic pattern but not in terms of the occurrence positions. Here, the inexactness could be due to some noise (replacement, insertion and deletion errors) introduced in the structure of the pattern which in turn disturbs the perfect periodic nature. The bound on the number of such errors allowed gives us the case of $(\kappa \neq 0, \gamma = 0)$ where κ is the maximum number of such errors allowed in a particular copy of the periodic pattern.

Following [62], approximate periodicity can be defined as follows: Given two strings T and P and a distance function δ, if there exists a partition of T into disjoint blocks of substrings, i.e., $T = P_1P_2P_3....P_x$ ($P_i \neq \varepsilon$, where ε is an empty string) such that $\delta(P, P_i) \leq k$ for $1 \leq i \leq x$ and $\delta(P', P_x) \leq k$, where P' is some prefix of P, we say that P is a k-approximate period of T (or P is an approximate period of T with distance k). This is the case of full segment periodicity where errors are allowed only in the structure of the pattern. Also, the strings P_i can be separated by a fixed length string Y, where Y is any string of fixed length q such that $Y \in \Sigma^q$. This leads to the case of partial periodicity where errors are allowed in the structure of the pattern. The variable k corresponds to the parameter κ. Sim et al. [62] discussed three problems related to the case of $(\kappa \neq 0, \gamma = 0)$. The problem of finding the minimum k for given strings T (of length n) and P (of length

m) and a distance function δ such that P is a k-approximate period of T can be solved in $O(mn^2)$ time when δ is a weighted edit distance function, in $O(mn)$ time when δ is a unit cost edit distance function, and in $O(n)$ time when δ is the Hamming distance. The second problem of finding a substring P of T such that P is an approximate period of T with the minimum distance, given T and a distance function δ can be solved in $O(n^4)$ time under weighted edit distance and in $O(n^3)$ time under relative Hamming distance. The third problem of finding a string P that is an approximate period of T with the minimum distance, given a string T and a relative distance function δ where P can be any string not necessarily a substring of T was shown to be NP-complete [62].

As an example of approximate segment periodicity, consider text T= $abcdefghij\ abcdefghij\ abcd\underline{\boldsymbol{d}}fg\underline{\boldsymbol{i}}ij\ abcdefghij\ ab\underline{\boldsymbol{e}}defghij$. Here, some occurrences of the pattern $P = abcdefghij$ have errors introduced in the structure, where the occurrence vector for the pattern with exact matching is $V = [0, 10, 30]$ and the occurrence vector for the pattern with approximate matching (with $\kappa = 2, \gamma = 0$) is $V = [0, 10, 20, 30, 40]$. But notice that in each case, the separation between the primitive patterns is always zero, thus $\gamma = 0$ since there in no variation in the spacing.

As an example of approximate partial periodicity, consider text T= $\boldsymbol{abcde}gdffg\ \boldsymbol{abcde}yiqas\ \boldsymbol{ab\underline{d}de}msmsx\ \boldsymbol{abcde}hqeoe\ \boldsymbol{ab\underline{d}d\underline{h}}rscdf$. Here, some occurrences of the pattern $P = abcde$ have errors introduced in the structure, where the occurrence vector for the pattern with exact matching is $V = [0, 10, 30]$ and the occurrence vector for the pattern with approximate matching (with $\kappa = 2, \gamma = 0$) is $V = [0, 10, 20, 30, 40]$. But notice that in each case, the separation between the primitive patterns is always 5, thus $\gamma = 0$ since there in no variation in the spacing.

The algorithm by Elfeky et al. [22] uses convolution based technique (CONV) to detect segment and symbol periodicity in $O(n \log n)$ time, and detects periods in the presence of replacement noise introduced in the structure of the pattern. As an improvement to CONV technique, Elfeky et al. [23] proposed a different algorithm WARP to also handle insertion and deletion noise introduced in the structure of the pattern. It requires time in $O(n^2)$, but it can only detect segment periodicity.

4.1.3 The case of $(\kappa = 0, \gamma \neq 0)$

The approximation is in terms of the occurrence positions of a periodic pattern. Here, the insertion and deletion noise introduced between the pattern occurrences perturb the perfect periodic nature as opposed to the previous case where the approximation is in terms of the structure of the periodic pattern due to the noise introduced in the structure of the pattern. In this case, substitution errors might be introduced between the occurrences of the pattern but it has no effect on the occurrence positions of the periodic pattern. The periodic pattern is allowed to drift away from its expected periodic positions up to a certain limit given by the parameter γ leading to the case of $(\kappa = 0, \gamma \neq 0)$.

To accommodate this type of variability, approximate periodicity can also be defined as follows: Let T be a string of length $n = |T|$. T is called periodic if $T = (PY)^i pref(P)$ where $i \in \mathbb{N}, i \geq 2$, P and Y are substrings of T such that $|PY| \leq n/2$, P is fixed and Y is a substring of T of length q, where the value of q can vary upto a certain limit (i.e., $q \pm \gamma$, $\gamma \geq 0$), and Y is any string such that $Y \in \Sigma^q$. The smallest such substring P is called the partial period of T.

For example, consider a text T= ***abcde****gdikl* ***abcde****yiqas* ***abcde****yiaso* ***abcde****heoe* ***abcde****rscdf* ***abcde****gttyur* ***abcde****rewqa*. Here, errors are introduced between the occurrences of the pattern, thus giving us the occurrence vector $V = [0, 10, 20, 30, 39, 49, 60]$, where the distance between adjacent primitive patterns deviate from the perfect period length of 10, i.e., 39-30=9, 60-49=11, etc. Notice that there are no errors introduced in the structure of the pattern itself, but the spacing between the occurrences varies from the exact period value of 10 due to errors introduced between the occurrences.

This kind of periodicity is also described as asynchronous periodicity in [32, 33, 49, 71, 72], where some perturbation is allowed between the repetitions of the pattern. The papers proposed solutions to mine periodic patterns in such cases. Rasheed et al. [57] developed an algorithm called STNR (Suffix Tree based Noise Resilient algorithm) which mines periodic patterns in the presence of noise where periodic occurrences are allowed to drift (shifted ahead or back) within an allowable limit (tt) which is similar to our parameter γ. The worst case time complexity for the STNR algorithm

is in $O(n^3)$.

4.1.4 The case of $(\kappa \neq 0, \gamma \neq 0)$

The approximation is in terms of both the occurrence positions and the structure of a periodic pattern. Here, errors or noise are allowed to occur *simultaneously* in both the primitive repeating pattern and in the separation distance between occurrences. That is, both κ and γ could be non-zero.

The cases with $(\kappa \neq 0, \gamma = 0)$ and $(\kappa = 0, \gamma \neq 0)$ can be combined to define another form of approximate periodicity as follows: Given two strings T and P and a distance function δ, T is called periodic if $T = (P_a Y)^i pref(P)$, where $i \in \mathbb{N}, i \geq 2$, such that $|P_a Y| \leq n/2$ and P_a is an approximate version of P ($P_a \neq \varepsilon$, where ε is an empty string) such that $\delta(P, P_a) \leq \kappa$, and Y is a substring of T of length q, where the value of q can vary upto a certain limit γ (i.e., $q \pm \gamma,\ \gamma \geq 0$). The smallest such substring P is called the approximate period of T.

For example, consider a text T= ***abcde***$gdaog$ ***abcde***$yiqas$ ***ab<u>d</u>de***$yuiqr$ ***abcde***$heoe$ ***<u>abce</u>***$rscdf$ ***abcde***$gttyur$ ***abcde***$rewqa$. Here, errors are introduced between the occurrences of the pattern and also in the structure of the pattern, thus giving us the occurrence vector $V = [0, 10, 30, 48, 59]$. Notice that certain occurrences are missed, but they can be detected by using approximate matching with errors allowed in the structure of the pattern. This will detect occurrences at positions 20 and 39 giving us the occurrence vector $V = [0, 10, 20, 30, 39, 48, 59]$. Still, distances between certain occurrence positions deviate from the perfect period value of 10. For instance, we have differences of 39-30=9, 48-39=9, 59-48=11, etc., which can be considered as approximation between the occurrence positions of the pattern.

While previous methods on approximate periodicity have considered the cases with $(\kappa = 0, \gamma \neq 0)$ [23,62], and $(\kappa \neq 0, \gamma = 0)$ [32,33,49,57,71,72], there is no published work that directly addresses the case of simultaneous errors in both the primitive pattern and their separation space (case of $\kappa \neq 0, \gamma \neq 0$).

4.2 A Method for Approximate Periodicity Mining ($\kappa \neq 0, \gamma \neq 0$)

Time series data represents events related to real-world phenomena such as stock market growth, transactions at a supermarket, hydrological data, power consumption, network traffic, weather data, etc. Repetitions are an inherent part of the data representing such phenomena. The goal of periodicity mining is to mine periods out of the time series data to provide us with valuable information about the repeating cycles that help us to understand the nature of the phenomena and also forecast future events. The time series data is prone to errors from different sources, e.g. data acquisition method, transient errors, or the errors can be an inherent part of the data itself because of which the periodic nature of the data represented by the string may be inexact [7]. However, it is important to mine the periods efficiently and effectively in spite of the presence of noise. This leads us to the problem of approximate periodicity mining.

The goal of periodicity mining is to find temporal regularities in the time series data. The frequency of periodicity for a particular periodic pattern is measured using a metric called ***confidence***. The confidence is defined as the ratio of the actual frequency of the pattern to the expected frequency within the whole or subsection of the time series. The presence of noise has a negative impact on this metric. A pattern P is said to be perfectly periodic with period p from the position of its first occurrence position ($stPos$), if it exactly occurs at positions $stPos + i * p$ ($i \in \mathbb{N}, i \geq 0$) in the text until its last occurrence [57]. Presence of noise in the data disturbs the perfectness, leading to imperfect periodicity. Noise can be due to replacement, insertion or deletion errors. The goal is to improve the confidence of periodicity detection by considering approximation simultaneously both in the occurrence positions and the structure of the periodic pattern itself. In [57], Rasheed et al. proposed STNR (Suffix Tree based Noise Resilient) algorithm for periodicity mining, which handles the case of ($\kappa = 0, \gamma \neq 0$). In this section we present STNR-A (Suffix Tree based Noise Resilient algorithm with Approximation), an improvement of STNR to handle the case of ($\kappa \neq 0, \gamma \neq 0$). Our method uses the CST to generate the occurrence vectors, and then analyzes the occurrences vectors to support inexact periodicities with ($\kappa \neq 0, \gamma \neq 0$).

4.2.1 STNR: Features of Existing Algorithm

STNR handles noise (insertion, deletion and replacement errors) introduced between the occurrences of the periodic pattern, but not the noise introduced in the structure of the pattern. The algorithm uses suffix tree to capture repeated substrings in the text. For each such string represented by an internal node's path label an occurrence vector (V) is generated. An occurrence vector gives the list of the starting positions in the text T, of all the exact occurrences of a given pattern P. Thus, the positions of patterns in which errors were introduced are not recorded. The periodicity mining algorithm then processes the occurrence vector to check if the pattern corresponding to the given occurrence vector is periodic. The algorithm considers each element ($V[j]$) of the occurrence vector as a potential start ($stPos$) of the periodic nature and the difference ($p = V[j+1] - V[j]$) between the adjacent elements as a potential period value, where ($0 \leq j \leq \eta_V$). It then scans the occurrence vector and increases the frequency count if the occurrence vector values are periodic w.r.t to $stPos$ and period value p [57]. The worst case time complexity of the algorithm is $O(n^3)$, with at most $n-1$ patterns, with an occurrence vector of size at most n generated for each such pattern and $O(n^2)$ time to process each occurrence vector, where n is the size of the time series data T. The current framework handles asynchronous periodicity ($\kappa = 0, \gamma \neq 0$) which is due to insertion and deletion noise between the occurrences of the pattern. Thus, the occurrences are allowed to drift from their expected positions up to an allowable limit (denoted by (tt) in [57]). The threshold (tt) of the drift is a pre-defined parameter for the STNR algorithm which is similar to the parameter γ defined in our taxonomy.

For example, consider a pattern P that is perfectly periodic with period value $p = 50$ within a subsection of time series with $stPos = 0$ and $endPos = 500$. SNTR only handles the cases of $\kappa = 0, \gamma \neq 0$. Figure 4.1 and Table 4.1 show the various cases of approximate periodicity handled by the STNR algorithm.

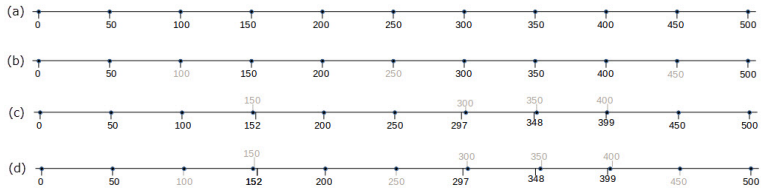


Figure 4.1: Different forms of approximate periodicity handled by the STNR algorithm. See also Table 4.1.

Table 4.1: Explanation of different forms of inexact periodicity shown in Figure 4.1.

Case (a): V=[0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500], η_V=11		
Perfect Periodicity	Confidence= 11/11= 1.0	No noise
Case (b): V = [0, 50, 150, 200, 300, 350, 400, 500], η_V=8		
Imperfect Periodicity ($\gamma = 0$)	Confidence = 8/11 = 0.73	Handles replacement noise by ignoring certain positions of expected occurrences (100, 250, 450). No insertion or deletion noise.
Case (c): V= [0, 50, 100, 152, 200, 250, 297, 348, 399, 450, 500], η_V=11		
Imperfect Periodicity ($\gamma = 3$)	Confidence= 11/11= 1.0	Noise between the occurrences of the pattern in the form of insertion and deletion errors. [Actual occurrence/Expected occurrence: 152/150, 200/202, 297/300, 348/347, 399/398, 450/449]
Case (d): V=[0, 50, 152, 200, 297, 348, 399, 500], η_V=8		
Imperfect Periodicity ($\gamma = 3$)	Confidence = 8/11 = 0.73	Handles insertion and deletion noise between the occurrences of the pattern with the use of γ. [Actual occurrence/Expected occurrence: 152/150 , 200/202 , 297/300, 348/347, 399/398, 500/599]. Handles replacement noise by ignoring certain positions of expected occurrences (250,449)

4.2.2 STNR-A: Periodicity Mining with Approximation

We propose an algorithm that simultaneously handles insertion, deletion and replacement noise introduced both between the occurrences of the pattern and in the structure of the pattern i.e., the case of ($\kappa \neq 0, \gamma \neq 0$). The algorithm employs approximate pattern matching to check for the occurrence of an approximate version of the pattern at expected positions of occurrence, for every scan of the occurrence vector. The result is more accurate with improved confidence in the periodicity detection for a given pattern.

Approach

Given a text T of length n representing the practical time series data, a suffix tree or a compressed suffix tree is built and then the tree structure is traversed to record the occurrence vectors for all the internal nodes in the tree. The occurrence vector for a given internal node contains the list of all the leaf nodes in the subtree rooted at that particular internal node, which represents the starting positions of all the exact occurrences of the substring (representing the path label of the internal node) in the text.

A distance-based algorithm is used to process the occurrence vectors one at a time, to check if the pattern corresponding to the occurrence vector is periodic. The algorithm considers each element $V[j]$ of the occurrence vector as a potential start position ($stPos$) of periodicity, where ($0 \leq j \leq \eta_V$), and η_V the size of the current occurrence vector. It assumes the difference $p = V[j+1] - V[j]$ as the potential period and then makes a linear scan through the occurrence vector to check for periodicity with period length p. It checks the elements of the occurrence vector to see if the elements are periodic with respect to the position $stPos$ and period length p.

Consider the running example as described earlier, with occurrence vector $V = [0, 50, 150, 200, 300, 350, 400, 500]$. During the first scan of the occurrence vector, the algorithm considers $stPos = V[0] = 0$ as the start of the periodicity and assumes $p = V[1] - V[0] = 50$ as the potential period and scans the occurrence vector to check for periodicity. We observe that

the expected occurrences 100, 250, 450 might not be recorded due to some errors introduced at copies of the pattern, resulting in periodicity with low confidence. During the next scan, $stPos = V[1] = 50$ is considered as the potential start of periodicity and $p = V[2] - V[1] = 100$ as the potential period value, and scans the vector to look for periodicitygmg with $p = 100$.

Our idea is to use approximate pattern matching for missed expected occurrences to improve the confidence. Given a missed expected occurrence position (x), we use approximate pattern matching to find positions of approximate occurrences of pattern P with edit distance $\leq \kappa$ in the substring of text $T^{'} = T[x-\gamma....x+m+\gamma+\kappa]$, where κ is the bound on the number of errors allowed in a particular occurrence of the pattern and γ is the bound on the allowable drift from its expected occurrence position. The left range of the substring $T^{'}$ is chosen as $(x - \gamma)$ since the occurrence position of the pattern can drift to the left of its expected position due to deletion noise (γ number of symbols deleted) introduced in the text between the position (x) and its previous occurrence. The right range of the substring $T^{'}$ is chosen as $(x+\gamma+\kappa)$ to handle the worst cases. The occurrence position of the pattern can drift to the right of its expected position due to insertion noise (γ number of symbols inserted) introduced in the text between the position (x) and its previous occurrence. Also, there can be errors introduced in the structure of the pattern (maximum of κ number of errors, insertion in the worst case). Any standard approximate matching algorithm can be used to identify the position in the substring $T^{'}$ of an approximate occurrence of pattern P with least edit distance. The use of standard dynamic programming will result in $O(m^2)$ time to check for each such possible position x. Alternately, the method by Landau and Vishkin [42] can be used to perform the check in $O(\kappa m)$ worst-case time for each expected occurrence position x.

Algorithm 3 STNR ($\kappa = 0, \gamma \neq 0$).

```
1: for each occurrence vector V of size η_V for pattern P do
2:     for j ← 0 to η_V − 2 do
3:         p ←record potential period starting at position j
4:         set count of periodic occurrences to 0
5:         for i ← j to η_V do
6:             calculate the difference between the current occurrence V[i]
                   and last possible periodic occurrence currStPos
7:             calculate the number of periodic jumps from last possible
                    periodic occurrence currStPos to current occurrence V[i]
8:             if drift of current occurrence from expected occurrence is
                within allowable limits (γ) then
9:                 update last possible periodic occurrence (currStPos) ←
                       current occurrence V[i]
10:                increment count
11:            end if
12:        end for
13:        calculate confidence for the current periodic section
14:        if confidence ≥ threshold then
15:            add p to period list
16:        end if
17:    end for
18: end for
```

The STNR algorithm (Algorithm 3 on page 64), when making a scan through the occurrence vector for a given period p, does not check for possible occurrences that could be missed due to the errors in the structure of the pattern at expected occurrences. Thus, when the number of periodic jumps *(line 7)* from the last possible periodic occurrence *currStPos* to current occurrence position is greater than 1, it is important to check for possible occurrences of approximate versions of pattern P at expected positions of occurrences.

The algorithm STNR [57] is modified to accommodate the case of errors introduced in the structure of the pattern. Algorithm 4 (page 65) shows our proposed method STNR-A. Essentially, this modifies the STNR algorithm by inserting procedures to handle the approximate periodicity with

($\kappa \neq 0, \gamma \neq 0$), between Lines 7 and 8 of Algorithm 3. The new codes are added between Lines 8 and 24 in Algorithm 4. When the number of periodic jumps from the last possible periodic occurrence ($currStPos$) to current occurrence ($V[i]$) is greater than 1, then a call to check for an approximate occurrence of pattern P around a possible region of expected occurrence is made. The check returns a boolean flag $found$ set to true if such an occurrence is found and also returns the position value. The last possible occurrence ($currStPos$) is updated with the value returned and also the count of periodic occurrences is increased. If there exists no such occurrence then it returns false, and in this case the $skipCount$ is incremented. The variable $skipCount$ keeps track of the number of consecutive missed occurrences and when the value exceeds $maxSkipCount$, which is the limit on the number of consecutive missed occurrences, and thus indicates the end of the current periodic section. This helps us in identifying periodic sections that occur only in a subsection of the time series data. After the if statement of the check to "$\#jumps > 1$", the loop counter is decremented, as still the current occurrence position $V[i]$ is not considered to be part of the current periodic series. The Appendix A (on Page 85) contains a program listing implementing the Algorithm 4.

Algorithm 4 STNR-A ($\kappa \neq 0, \gamma \neq 0$).

```
1: for each occurrence vector V of size η_V for pattern P do
2:     for j ← 0 to η_V − 2 do
3:         p ←record potential period starting at position j
4:         set count of periodic occurrences to 0
5:         for i ← j to η_V do
6:             calculate the difference between the current occurrence V[i]
                   and last possible periodic occurrence currStPos
7:             calculate the number of periodic jumps from last possible
                       periodic occurrence currStPos to current occurrence V[i]
```

```
8:              if # such jumps > 1 then
9:                  check for an approximate occurrence of P around allowable
                        limits (γ) from the expected occurrence (currStPos+p)
                        and get its position if its found
10:                 if found =true then
11:                     increment count
12:                     update last possible occurrence (currStPos) with the
                            position found
13:                     set skipCount ← 0
14:                 else
15:                     update last possible occurrence (currStPos) to current
                            expected occurrence (currStPos + p)
16:                     increment skipCount
17:                     if skipCount > allowable limit on number of skips
                        (maxSkipCount) then
18:                         mark the end of current periodic section
19:                         exit the current loop
20:                     end if
21:                 end if
22:                 decrement loop counter /* as current occurrence V[i] is not
                            yet considered part of periodic occurrences */
23:                 continue
24:             end if
25:             if drift of current occurrence from expected occurrence is
                 within allowable limits (γ) then
26:                 update last possible periodic occurrence (currStPos) ←
                        current occurrence V[i]
27:                 increment count
28:             end if
29:         end for
30:         calculate confidence for the current periodic section
31:         if confidence ≥ threshold then
32:             add p to period list
33:         end if
34:     end for
35: end for
```

Algorithm Analysis

A suffix tree contains at most $2n-1$ nodes and always n leaves. Thus, there will be a maximum of $n-1$ internal nodes and an occurrence vector for each internal node. Let $\eta_V (\leq n)$, be the maximum size of any given occurrence vector. The algorithm considers every element $V[j]$ of the occurrence vector as a potential start of periodicity and $p = V[j+1] - V[j]$ as the potential period, where $0 \leq j \leq \eta_V$, and $\eta_V = |V|$. For a given value of j, the algorithm scans the occurrence vector to check if there exists a periodic pattern with current period p. The time required to make one such scan is $O(\eta_V)$ and the time required to process an occurrence vector of size at most η_V, is $O(\eta_V^2)$, without approximation.

Let us estimate the time required to perform periodicity mining with approximation. Let η_m be the number of possbile missed occurrences that are possible for a given pattern P and an occurrence vector V. The algorithm makes η_m number of checks for approximate matches while processing the occurrence vector. However, the value of η_m depends on the nature of data which is prone to errors. The maximum time required to make such checks could be $O(\kappa n)$ which is the time taken to record all the κ-approximate matches for a given pattern in the text T. Thus, the time required to process an occurrence vector with approximation, for a given pattern P is $O(\eta_V^2 + \kappa n)$. The time taken to process all the occurrence vectors (at most n) is $O(n * (\eta_V^2 + \kappa n)) \leq O(n^3)$, where n is the size of the text T, η_V is the maximum size of any occurrence vector and κ is the bound on the number of errors allowed in the structure of the pattern.

4.2.3 Preliminary Experimental Results

In this section, we present the comparison of periodicity results generated by algorithms STNR [57] and STNR-A. We generate synthetic data the way it has been done in [22, 57].

Experiment 1: Using synthetic data with periodic nature as described in Table 4.1 from Section 4.2.1, below are the results obtained.

Case a: V=[0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500], $\eta_V = 11$					
STNR	p=50	stPos=0	endPos=500	conf=1.0	count=11
STNR-A	p=50	stPos=0	endPos=500	conf=1.0	count=11
Case b: V=[0, 50, 150, 200, 300, 350, 400, 500], $\eta_V = 8$					
STNR	p=50	stPos=0	endPos=500	conf=0.73	count=8
STNR-A	p=50	stPos=0	endPos=500	conf=1.0	count=11
Case c: V=[0, 50, 100, 152, 200, 250, 297, 348, 399, 450, 500], $\eta_V = 11$					
STNR	p=50	stPos=0	endPos=500	conf=1.0	count=11
STNR-A	p=50	stPos=0	endPos=500	conf=1.0	count=11
Case d: V=[0, 50, 152, 200, 297, 348, 399, 500], $\eta_V = 8$					
STNR	p=50	stPos=0	endPos=500	conf=0.73	count=8
STNR-A	p=50	stPos=0	endPos=500	conf=1.0	count=11

Table 4.2: Comparison of STNR and STNR-A for cases defined in Table 4.1.

Observations: $(\kappa = 2, \gamma = 3))$

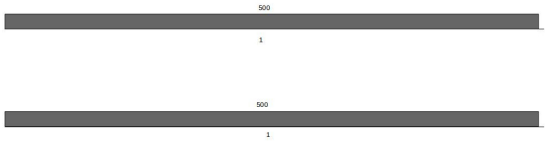
- **Case a** $(\kappa = 0, \gamma = 0))$**:** In the case of perfect periodicity, both algorithms produced the same result.
- **Case b** $(\kappa \neq 0, \gamma = 0))$**:** The case of imperfect periodicity, where there are certain occurrences missed (100,250,450) due to errors in the pattern. The STNR-A has better confidence as compared to SNTR due to approximate matching employed in the case of missed occurrences. For the given example, there exists approximate occurrences of the pattern at all the missed occurrence positions.
- **Case c** $(\kappa = 0, \gamma \neq 0))$**:** In the case of imperfect periodicity where errors are between occurrences of periodic pattern, the algorithms produced the same result.
- **Case d** $(\kappa \neq 0, \gamma \neq 0))$**:** The case of imperfect periodicity where there are certain occurrences missed (100,250,449) due to errors in the pattern and also drift of occurrences (152:drift=+2, 200:drift=-2, 297:drift=-3, 348:drift=+1, 399:drift=+1, 500:drift=+1) due to errors between the occurrences of the pattern. The STNR-A has better confidence as compared to SNTR due to approximate matching employed

in the case of missed occurrences. For the given example, there exists approximate occurrences of the pattern at all the missed occurrence positions.

Experiment 2: We generated synthetic data for a pattern $P = abcde$ with $p = 20$ and $|\Sigma| = 10$, with perfect periodic nature. And then introduce various degrees of noise and compare the results from both algorithms.

Values of Parameters: $\kappa = 3,\ \gamma = 3,\ maxSkipCount = 3,\ conf\_threshold = 0.5,\ dmax = p * maxSkipCount$

The results are given in the format: $P, p, (N)[stPos, endPos, conf, count]$, where P is the periodic pattern, p the period length, (N) the number of periodic subsections detected, $[stPos, endPos, conf, count]$ represents the starting position, end position, confidence and count of periodic occurrences for the given pattern P with period p. Multiple sets of these values indicate the pattern is found with the same period in multiple subsections of the text whose start and end values are specified by $stPos$ and $endPos$. $RID\ noise$ refers to replacement, insertion, and deletion noise ratio, and run#i represents the i-th run of the experiment.



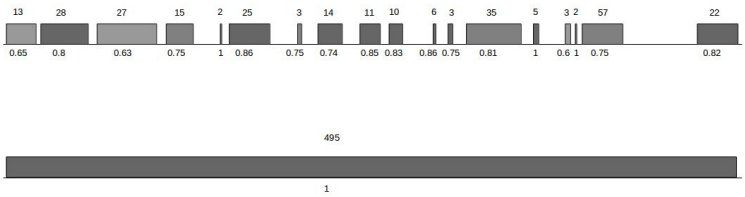
$P : abcde,\ p : 20$
STNR: (1) [0, 9980, 1, 500]

STNR-A: (1) [0, 9980, 1, 500]

Figure 4.2: Comparison of STNR and STNR-A in the case of perfect periodicity

Perfect Periodicity, RID noise=0.0: $n = 10,000,\ p = 20,\ |\Sigma| =$

10. The results of both STNR and STNR-A as shown in Figure 4.2 are the same, in the case of perfect periodicity.



$P : abcde,\ p : 20$

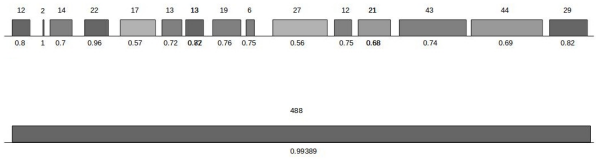
STNR: (18) [95, 500, 0.65, 13], [561, 1242, 0.8, 28], [1324, 2154, 0.63, 27], [2256, 2638, 0.75, 15], [2987, 3007, 1, 2], [3107, 3662, 0.86, 25], [4033, 4093, 0.75, 3], [4312, 4668, 0.74, 14], [4874, 5116, 0.85, 11], [5274, 5494, 0.83, 10], [5868, 5990, 0.86, 6], [6068, 6130, 0.75, 3], [6324, 7188, 0.81, 35], [7224, 7304, 1, 5], [7651, 7728, 0.6, 3], [7790, 7810, 1, 2], [7871, 9373, 0.75, 57], [9434, 9980, 0.81, 22]

STNR-A: (1) [95, 9980, 1, 495]

Figure 4.3: Comparison of STNR and STNR-A with RID=0.1 run# 1

RID noise=0.1, run#1: $n = 10,000,\ p = 20,\ |\Sigma| = 10$. As shown in Figure 4.3, it is observed that STNR outputs multiple periodic sections for the same pattern and same period value, while STNR-A outputs a single subsection.

RID noise=0.1, run#2: $n = 10,000,\ p = 20,\ |\Sigma| = 10$. As shown in Figure 4.4, it is observed that STNR outputs multiple periodic sections for the same pattern and same period value, while STNR-A outputs a single subsection.

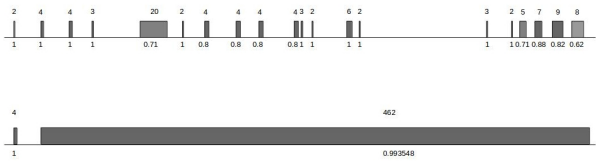


$P : abcde,\ p : 20$

STNR: (15) [184, 488, 0.8, 12], [708, 728, 1, 2], [828, 1208, 0.7, 14], [1412, 1849, 0.96, 22], [2008, 2600, 0.59, 17], [2721, 3062, 0.72, 13], [3124, 3429, 0.87, 13], [3587, 4066, 0.76, 19], [4149, 4291, 0.75, 6], [4607, 5537, 0.56, 27], [5652, 5948, 0.75, 12], [6052, 6647, 0.68, 21], [6747, 7887, 0.74, 43], [7969, 9200, 0.7, 44], [9296, 9980, 0.83, 29]

STNR-A: (1) [184, 9980, 0.994, 488]

Figure 4.4: Comparison of STNR and STNR-A with RID=0.1 run# 2



$P : abcde,\ p : 20$

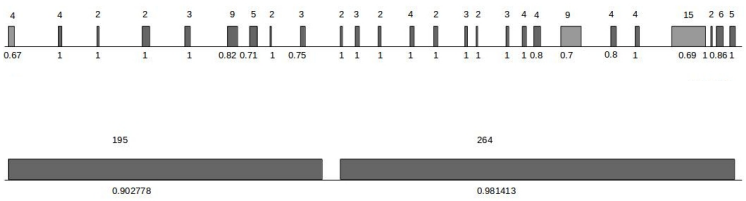
STNR: (20) [208, 228, 1, 2], [661, 717, 1, 4], [1142, 1206, 1, 4], [1530, 1568, 1, 3], [2350, 2887, 0.71, 20], [3065, 3085, 1, 2], [3445, 3521, 0.8, 4], [3971, 4049, 0.8, 4], [4350, 4430, 0.8, 4], [4954, 5036, 0.8, 4], [5071, 5110, 1, 3], [5254, 5274, 1, 2], [5838, 5935, 1, 6], [6048, 6068, 1, 2], [8210, 8250, 1, 3], [8627, 8647, 1, 2], [8763, 8881, 0.71, 5], [9022, 9158, 0.88, 7], [9319, 9521, 0.82, 9], [9640, 9879, 0.62, 8]

STNR-A: (2) [208, 268, 1, 4], [661, 9940, 0.993, 462]

Figure 4.5: Comparison of STNR and STNR-A with RID=0.2 run# 1

RID noise=0.2, run#1: $n = 10,000,\ p = 20,\ |\Sigma| = 10$. As shown in Figure 4.5, it is observed that STNR outputs multiple periodic sections for the same pattern and same period value, while STNR-A outputs only two subsections.

RID noise=0.2, run#2: $n = 10,000,\ p = 20,\ |\Sigma| = 10$. As shown in Figure 4.6, it is observed that STNR outputs multiple periodic sections for the same pattern and same period value, while STNR-A outputs only two subsections.



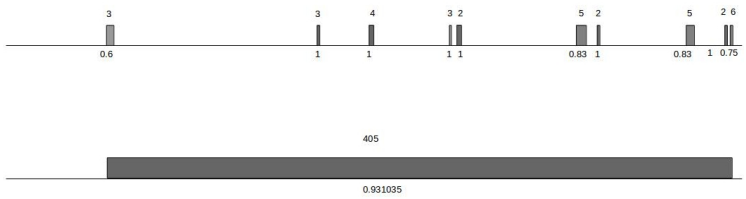
$P : abcde,\ p : 20$

STNR: (26) [102, 200, 0.67, 4], [783, 843, 1, 4], [1306, 1326, 1, 2], [1919, 1939, 1, 2], [2491, 2531, 1, 3], [3071, 3295, 0.82, 9], [3376, 3496, 0.71, 5], [3652, 3672, 1, 2], [4057, 4117, 0.75, 3], [4604, 4624, 1, 2], [4807, 4846, 1, 3], [5119, 5139, 1, 2], [5543, 5606, 1, 4], [5864, 5884, 1, 2], [6283, 6324, 1, 3], [6440, 6460, 1, 2], [6844, 6883, 1, 3], [7066, 7126, 1, 4], [7225, 7327, 0.8, 4], [7586, 7826, 0.69, 9], [8267, 8344, 0.8, 4], [8597, 8657, 1, 4], [9101, 9517, 0.68, 15], [9618, 9638, 1, 2], [9699, 9819, 0.86, 6], [9880, 9960, 1, 5]

STNR-A: (2) [102, 4373, 0.903, 195], [4604, 9960, 0.98, 264]

Figure 4.6: Comparison of STNR and STNR-A with RID=0.2 run# 2

RID noise=0.3, run#1: $n = 10,000,\ p = 20,\ |\Sigma| = 10$. As shown in Figure 4.7, it is observed that STNR outputs multiple periodic sections for the same pattern and same period value, while STNR-A outputs a single subsection.



$P : abcde,\ p : 20$
STNR: (10) [1367, 1447, 0.6, 3], [4226, 4265, 1, 3], [4929, 4992, 1, 4], [6018, 6059, 1, 3], [6121, 6141, 1, 2], [7745, 7844, 0.83, 5], [8025, 8045, 1, 2], [9242, 9341, 0.83, 5], [9757, 9777, 1, 2], [9839, 9980, 0.75, 6]

STNR-A: (1) [1367, 9980, 0.931, 405]

Figure 4.7: Comparison of STNR and STNR-A with RID=0.3 run# 1

It is observed that with the increasing noise, the STNR produces more number of subsections with decreased confidence, which is due to the noise introduced in the structure of the pattern, whereas, the STNR-A detects those occurrences where the errors are in the structure of the pattern and thus reporting less number of subsections and with improved confidence. It is clearly observed that STNR-A outperforms the algorithm STNR for efficient and effective mining of periods in the presence of noise in both the structure of the periodic pattern and also between the occurrences of the periodic pattern.

4.2.4 Improved practical time for long patterns

The time involved in the checks for an occurrence of an approximate version of the periodic pattern at expected positions will be costly if the size of the pattern is large. The goal is to reduce the overall time involved in approximate matching for periodicity detection in the case of long patterns by using a two-phase hypothesis-generation and hypothesis-verification approach using

approximate q-gram filtering.

Hypothesis Generation Phase: Also called the filtering phase, is used to identify regions of text which could be potential matches. First, either the text or pattern is partitioned into consecutive regions of a given predefined length called q-grams. Efficient exact matching algorithms will be employed to detect length-q intervals of T which could potentially be in approximate occurrences with P, called the surviving intervals. The goal is to eliminate, as many as possible, the other regions of T, called the non-surviving intervals, which definitely cannot contain approximate occurrences of P. The end result of this phase is a vector $\mathcal{I}$ which contains the starting positions in T of exact occurrences of any q-length region of P.

Hypothesis verification Phase: In this phase, for each hypothesis generated i.e., for each $i \in \mathcal{I}$, some approximate matching method is employed explicitly between a sufficiently large substring of T around position i and the pattern P to check if there is an approximate occurrence of P in a sufficiently large interval around i.

The different methods based on this approach can vary depending on their choice of the string to partition, exact matching algorithms to be employed in the hypothesis generation phase, the choice of value for q and the choice of the definition of the region. The time cost for the verification phase is generally high, and the overall performance of the algorithm depends on the efficiency of the hypothesis generation phase, in terms of the time needed for the generation phase and also on the number of hypotheses generated [1]. The key point is that there needs to be a balance between the hypothesis generation and hypothesis verification phases, as the reduction in the time consumed in one phase causes an increase in the time consumed in the other phase [29].

Following Baeza-Yates et al. [12], the pattern P in partitioned into consecutive (non-overlapping) regions called q-grams each of length q, where $q = \lfloor \frac{m}{\kappa+1} \rfloor$ such that there will be $(\kappa + 1)$ regions each of full-length q and the last region can be of length less than q. The basis is the fact that even if the κ-errors are distributed in each of the separate κ regions, there will be a $(\kappa + 1)^{th}$ region which does not contain any errors and will exactly match with one q-length region of $T^{'}$, a substring of T.

Using the suffix tree of T, we can perform hypothesis generation by searching in text T, for all the exact occurrences of each of the (κ+1) substrings of P, each of length q. The time taken during the search phase to find the exact occurrence positions of all the q-grams will be proportional to

$$O((\kappa + 1) * q * \log(\Sigma) + \eta_h) \simeq O(m * \log(\Sigma) + \eta_h)$$

where $\eta_h = |\mathcal{I}|$ is the number of the exact occurrences of all of the q-grams. The list of the exact occurrences of all of the q-grams will be maintained in the vector $\mathcal{I}$ in sorted order.

During the scan of the occurrence vectors, in the case of long patterns instead of employing approximate matching for every possible missed expected occurrence position (x), we first check to see if there exists a possible approximate match around the position x. To check for an approximate occurrence of the pattern at given expected positions (x), we make a linear pass through the vector $\mathcal{I}$. For example, let x be a particular expected occurrence position, we apply approximate pattern matching only if x exists in any subsection $T[i - m - \kappa...i + m + \kappa]$, where ($i \in \mathcal{I}$) and κ is the bound on the number of errors allowed in the structure of the pattern. Given $\mathcal{I}$, this check can be performed in constant time for each position x in T or $O(n)$ time for all the checks for each pattern. The worst case time involved in processing an occurrence vector for a given pattern will still remain $O(\eta_V^2 + \kappa n)$, however, the practical time will be reduced as some approximate matching checks can be eliminated with the use of hypothesis generation and hypothesis verification steps.

Chapter 5

Conclusions

Periodicity mining has gained a lot of importance due to its varied applications such as in weather forecasting, stock market prediction and analysis, analysing patterns of power consumption, etc. However, there are challenges involved in mining meaningful periods out of the practical time series data. Processing large amounts of data and handling noise which is an inherent part of the data, make efficient and effective mining a challenging problem. In this work, we propose the use of compressed suffix trees for efficient periodicity mining to handle large amounts of data. Though suffix trees can efficiently capture repeated substrings in the text, its high practical space requirements make it difficult to perform periodicity mining in main memory, especially for large data. With a comprehensive empirical analysis on the practical usage of the suffix data structures for periodicity mining, we conclude that the compressed suffix trees can replace suffix trees for space-efficient periodicity mining.

It is important to mine periods in spite of the noise present in the practical time series data, as it gives accurate information about the temporal regularities in the time series data. The noise can occur simultaneously both between the occurrences of a periodic pattern and in the structure of the pattern itself which leads us to the problem of approximate periodicity. We provide a taxonomy for approximate periodicity using our two defined parameters κ and γ, where κ is the bound on the number of errors allowed in the structure of the periodic pattern and γ is the bound on the number

of errors allowed between the periodic occurrences of the pattern. We propose an improved algorithm for approximate periodicity mining. Our new algorithm STNR-A runs in time $O(\eta_V^2 + \kappa n)$ and handles the extreme case of $(\kappa \neq 0, \gamma \neq 0)$, where η_V is the maximum size of the occurrence vector and n the length of the time series data. The results from preliminary experiments using synthetic data, show that the algorithm can detect periods accurately with improved confidence, which otherwise could go unidentified due to the noise in the data.

Bibliography

[1] D. Adjeroh, A. Mukherjee, T. Bell, M. Powell, and Nan Zhang. Pattern matching in BWT-transformed text. In *Data Compression Conference, 2002. Proceedings. DCC 2002*, pages 445–, 2002.

[2] Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Science & Business Media, 2008.

[3] Donald Adjeroh and Fei Nan. Suffix-Sorting via Shannon-Fano-Elias Codes. *Algorithms*, 3(2):145–167, 2010.

[4] Akram Al-Rawi, Azzedine Lansari, and Faouzi Bouslama. A new non-recursive algorithm for binary search tree traversal. In *Electronics, Circuits and Systems, 2003. ICECS 2003. Proceedings of the 2003 10th IEEE International Conference on*, volume 2, pages 770–773. IEEE, 2003.

[5] Srinivas Aluru. Suffix trees and suffix arrays. *Handbook of Data Structures and Applications*, 2004.

[6] Amihood Amir, Alberto Apostolico, Gad M Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range lcp. *Journal of Computer and System Sciences*, 80(7):1245–1253, 2014.

[7] Amihood Amir, Estrella Eisenberg, and Avivit Levy. Approximate periodicity. In *Algorithms and Computation*, pages 25–36. Springer, 2010.

[8] Amihood Amir and Avivit Levy. Approximate period detection and correction. In *String Processing and Information Retrieval*, pages 1–15. Springer, 2012.

[9] Alberto Apostolico and Raffaele Giancarlo. The boyer-moore-galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986.

[10] Alberto Apostolico and Franco P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22(3):297–315, 1983.

[11] Ricardo Baeza-Yates and Gaston H Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[12] Ricardo A Baeza-Yates and Chris H Perleberg. Fast and practical approximate string matching. In *Combinatorial Pattern Matching*, pages 185–192. Springer, 1992.

[13] Christos Berberidis, Walid G Aref, Mikhail Atallah, Ioannis Vlahavas, Ahmed K Elmagarmid, et al. Multiple and partial periodicity mining in time series databases. In *ECAI*, volume 2, pages 370–374, 2002.

[14] Jean Berstel. Fibonacci words—a survey. In *The book of L*, pages 13–27. Springer, 1986.

[15] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[16] Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *Experimental Algorithms*, pages 94–105. Springer, 2010.

[17] Christian Charras and Thierry Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.

[18] Pizza&Chili Corpus and Compressed Indexes. their testbeds, 2005.

[19] Max Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[20] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[21] Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74(5):796–807, 2008.

[22] Mohamed G Elfeky, Walid G Aref, and Ahmed K Elmagarmid. Periodicity detection in time series databases. *Knowledge and Data Engineering, IEEE Transactions on*, 17(7):875–887, 2005.

[23] Mohamed G Elfeky, Walid G Aref, and Ahmed K Elmagarmid. Warp: Time warping for periodicity detection. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

[24] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[25] Simon Gog. *Compressed suffix trees: Design, construction, and applications*. PhD thesis, PhD thesis, Univ. of Ulm, Germany, 2011.

[26] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[27] Simon Gog and Johannes Fischer. Advantages of shared data structures for sequences of balanced parentheses. In *Data Compression Conference (DCC), 2010*, pages 406–415. IEEE, 2010.

[28] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[29] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[30] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 106–115. IEEE, 1999.

[31] Jiawei Han, Wan Gong, and Yiwen Yin. Mining segment-wise periodic patterns in time-related databases. In *KDD*, pages 214–218, 1998.

[32] Kuo-Yu Huang and Chia-Hui Chang. Mining periodic patterns in sequence data. In *Data Warehousing and Knowledge Discovery*, pages 401–410. Springer, 2004.

[33] Kuo-Yu Huang and Chia Hui Chang. Smca: a general model for mining asynchronous periodic patterns in temporal databases. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):774–785, 2005.

[34] Costas S Iliopoulos, Dennis Moore, and William F Smyth. A characterization of the squares in a fibonacci string. *Theoretical Computer Science*, 172(1):281–291, 1997.

[35] Piotr Indyk, Nick Koudas, and S Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *VLDB*, pages 363–372, 2000.

[36] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

[37] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1988. AAI8918056.

[38] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[39] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[40] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[41] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Combinatorial Pattern Matching*, pages 200–210. Springer, 2003.

[42] Gad M Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 220–230. ACM, 1986.

[43] Gad M Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

[44] Jie Lin and Don Adjeroh. All-against-all circular pattern matching. *The Computer Journal*, 55(7):897–906, 2012.

[45] Jie Lin, Yue Jiang, and Don Adjeroh. Circular pattern discovery. *The Computer Journal*, 2014.

[46] Sheng Ma and Joseph L Hellerstein. Mining partially periodic event patterns with unknown periods. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 205–214. IEEE, 2001.

[47] Michael G Main and Richard J Lorentz. An o (n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[48] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[49] Fahad Maqbool, Shariq Bashir, and A Rauf Baig. E-map: Efficiently mining asynchronous periodic patterns. *International Journal of Computer Science and Network Security*, 6(8A):174–179, 2006.

[50] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[51] J Ian Munro, Venkatesh Raman, and S Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[52] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[53] Gonzalo Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.

[54] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[55] Enno Ohlebusch, Johannes Fischer, and Simon Gog. Cst++. In *String Processing and Information Retrieval*, pages 322–333. Springer, 2010.

[56] Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *String Processing and Information Retrieval*, pages 51–62. Springer, 2009.

[57] Faraz Rasheed, Mohammed Alshalalfa, and Reda Alhajj. Efficient periodicity mining in time series databases using suffix trees. *Knowledge and Data Engineering, IEEE Transactions on*, 23(1):79–94, 2011.

[58] Luís MS Russo, Gonzalo Navarro, and Arlindo L Oliveira. Fully-compressed suffix trees. In *LATIN 2008: Theoretical Informatics*, pages 362–373. Springer, 2008.

[59] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[60] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[61] Peter H Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms*, 1(4):359–373, 1980.

[62] Jeong Seop Sim, Kunsoo Park, Costas S Iliopoulos, and William F Smyth. Approximate periods of strings. In *Combinatorial Pattern Matching*, pages 123–133. Springer, 1999.

[63] Bill Smyth. Computing patterns in strings. 2003.

[64] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.

[65] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[66] Niko Välimäki, Veli Mäkinen, Wolfgang Gerlach, and Kashyap Dixit. Engineering a compressed suffix tree implementation. *Journal of Experimental Algorithmics (JEA)*, 14:2, 2009.

[67] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[68] Sun Wu and Udi Manber. Agrep–a fast approximate pattern-matching tool. *Usenix Winter 1992*, pages 153–162, 1992.

[69] Kostas F Xylogiannopoulos, Panagiotis Karampelas, and Reda Alhajj. Periodicity data mining in time series using suffix arrays. In *Intelligent Systems (IS), 2012 6th IEEE International Conference*, pages 172–181. IEEE, 2012.

[70] Jiong Yang, Wei Wang, and Philip S Yu. Infominer+: mining partial periodic patterns with gap penalties. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 725–728. IEEE, 2002.

[71] Jiong Yang, Wei Wang, and Philip S. Yu. Mining asynchronous periodic patterns in time series data. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):613–628, 2003.

[72] Jieh-Shan Yeh and Szu-Chen Lin. A new data structure for asynchronous periodic pattern mining. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 426–431. ACM, 2009.

[73] Shlomo Yona and Dotan Tsadok. Ansi c implementation of a suffix tree, 2003.

Appendices

A STNR-A: Algorithm Description

In the algorithm below, the variable p represents the current potential period, the variable $stPos$ holds the starting position in T of the current periodic series corresponding to the period p, and the variable $endPos$ represents the last position of the occurrence of the pattern P in the text T. The variable $currStPos$ is a moving reference which holds the value of the last possible periodic occurrence position and the variable $preOccur$ holds the value of the last valid periodic occurrence. The variable $count$ is incremented for every periodic position found. The variable $sumPer$ holds the summation of the period differences observed during the scan of an occurrence vector that helps in calculating the mean period length for the current periodic series. The variable $skipCount$ keeps track of the number of consecutive missed periodic occurrences.

The variable A represents the distance between the current occurrence position $V[i]$ and the current reference starting position ($currStPos$) [57], variable B represents the number of periodic jumps between the current occurrence position $V[i]$ and the current reference position ($currStPos$). The value of B greater than 1 indicates possible missed occurrences due to errors introduced in the structure of the pattern. Thus when ($B > 1$), there is a need to check for an approximate version of the pattern around expected position of occurrence ($currStPos+p$) i.e., p positions away from current reference starting position $currStPos$, by making a call to the method $kApproxCheck(currStPos + p, \gamma, \kappa)$. The method call returns two values a boolean $flag$ and a position value $aPos$. The $flag$ is set to $true$, if an approximate occurrence of pattern P is found in possible region around $currStPos + p$ and $aPos$ is set to the starting position in text T of such approximate occurrence found with least possible edit distance $\leq \kappa$. The $flag$ is set to $false$ otherwise. Thus, if the $flag$ is set to $true$, the count of periodic occurrences is incremented by 1 and the current reference position ($currStPos$) and last valid occurrence ($preOccur$) are updated with the value ($aPos$) returned by the method call. If the $flag$ is $false$, then the current reference position is updated to $currStPos + p$, to prevent the check for an

approximate occurrence at this position again in the cases of $B \geq 2$ and also the *skipCount* is incremented to account for missed occurrence positions. In the case of $B > 1$ the variable i is decremented to allow the current vector element $V[i]$ to be considered during the periodic checks to compensate for the increment of variable i in the increment part of the for loop. For $B \not> 1$, there exists no possible missed occurrences, and the variable C represents the drift of current vector element $V[i]$ value from the expected occurrence position. If the value of variable C falls within the value γ it is considered to be part of the current periodic series and the variable *count* is incremented.

In practical time series data the periods exits in either the whole time series data or only in a subsection of the data. To handle the later case, we define a parameter *maxSkipCount* which is the number of allowed consecutive missed occurrences that determines the end of a given periodic section. For example, consider an occurrence vector $V = [0, 50, 100, 350, 400, 450, 500]$ and $maxSkipCount = 3$, here there exists positions $150, 200, 250, 300$, which are missed occurrence positions. The missed occurrences could be due to the noise introduced in the pattern. While making a scan through the occurrence vector with potential period $p = 50$, the algorithm checks for approximate occurrences of the pattern at these positions. In case, the checks for approximate occurrences return false for all these positions, then the $(skipCount = 4) > maxSkipCount$, and thus the algorithm marks the end of one periodic section **(lines 24:27)** with $stPos = 0$ and $endPos = 100$.

Algorithm 1 STNR-A

1: **for** each occurrence vector (V) of size η_V for pattern P **do**
2: **for** $j \leftarrow 0$ **to** $\eta_V - 2$ **do**
3: $p \leftarrow V[j+1] - V[j]$;
4: $stPos \leftarrow V[j]$;
5: $endPos \leftarrow V[\eta_V - 1]$;
6: $currStPos \leftarrow stPos; preOccur \leftarrow -5$;
7: $count \leftarrow 0; sumPer \leftarrow 0; skipCount \leftarrow 0$;
8: **for** $i \leftarrow j$ **to** η_V **do**
9: $A \leftarrow V[i] - currStPos$;
10: $B \leftarrow Round(A/p)$;
11: **if** $(B > 1)$ **and** $((currStPos + p) < (n - p))$ **then**
12: $flag \leftarrow$ false;
13: $< flag, aPos > \leftarrow$ ***kApproxCheck(currStPos+p, γ, κ);***
14: **if** $flag$=true **then**
15: $sumPer \leftarrow sumPer + aPos - currStPos$;
16: $currStPos \leftarrow aPos$;
17: $preOccur \leftarrow aPos$;
18: $incrementcount(p)$;
19: $skipCount \leftarrow 0$;
20: **else**
21: $currStPos \leftarrow currStPos + p$;
22: $skipCount++$;
23: **if** $(skipCount > maxSkipCount)$ **then**
24: $endPos \leftarrow currStPos - skipCount * p$;
25: $break(exit\ \ current\ \ for\ \ loop)$;
26: **end if**
27: **end if**
28: $i--$;
29: **continue;**
30: **end if**
31: $skipCount \leftarrow 0$;
32: $C \leftarrow A - (p * B)$;
33: **if** $((-\gamma \leq C \leq \gamma)$ AND $(Round((preOccur - currSTPos)/p) \neq B))$ **then**
34: $currStPos \leftarrow V[i]$;
35: $preOccur \leftarrow V[i]$;
36: $incrementcount(p)$;
37: $sumPer \leftarrow sumPer + (p + C)$;
38: **end if**
39: **end for**
40: $meanP \leftarrow \frac{sumPer - p}{count(p) - 1}$
41: $conf(p) \leftarrow \frac{count(p)}{Perfect_Periodicity(p, stPos, X)}$
42: **if** $conf(p) \geq threshold$ **then**
43: add p to the period list;
44: **end if**
45: **end for**
46: **end for**

B Varying Period Value (p)

Table 1 shows the LCP values and entropy information for synthetic data with varying period value(p). The LCP values remain constant for varying period value, provided the alphabet size($|\Sigma|$=10) and data size(n = 1 MB) remain constant. The H_0 and H_1 entropy values increase with increasing period value.

p	#numNodes	#numNodes/n	maxLCP	meanLCP	maxLCP/n (X 10^{-3})	meanLCP/n(X 10^{-3})	H_0	H_1
10	2,097,146	1.999994	1,048,566	524,278	999.990	499.990	2.612	0.670
20	2,097,141	1.999989	1,048,556	524,268	999.981	499.981	2.998	1.184
30	2,097,134	1.999983	1,048,546	524,258	999.971	499.971	3.108	1.572
40	2,097,127	1.999976	1,048,536	524,248	999.962	499.962	3.124	1.871
50	2,097,121	1.999971	1,048,526	524,238	999.952	499.952	3.198	2.040
60	2,097,115	1.999965	1,048,516	524,228	999.943	499.943	3.225	2.177
70	2,097,109	1.999959	1,048,506	524,218	999.933	499.933	3.227	2.281
80	2,097,103	1.999953	1,048,496	524,208	999.924	499.924	3.208	2.452
90	2,097,096	1.999947	1,048,486	524,198	999.914	499.914	3.236	2.527
100	2,097,093	1.999944	1,048,476	524,188	999.905	499.905	3.236	2.543

Table 1: LCP, entropy and other attributes for synthetic data with varying period value(p)

CST_SADA								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
p	0.771	0.945	-1.000	-1.000	0.953	-0.903	0.161	-0.999
p	0.899	1.000	-1.000	-1.000	0.867	-1.000	0.200	-1.000

Table 2: Correlation values for CST_SADA using synthetic data with varying period value(p)

CST_SCT3								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
p	0.771	0.945	-1.000	-1.000	0.826	0.922	-0.603	-0.999
p	0.899	1.000	-1.000	-1.000	0.733	0.828	-0.556	-1.000

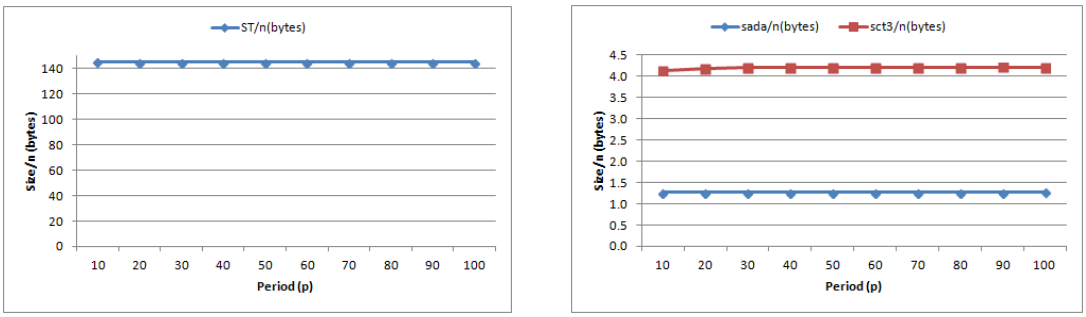
Table 3: Correlation values for CST_SCT3 using synthetic data with varying period value(p)

ST								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
p	0.771	0.945	-1.000	-1.000	-0.999	-0.713	-0.123	-0.999
p	0.899	1.000	-1.000	-1.000	-1.000	-0.764	0.000	-1.000

Table 4: Correlation values for ST using synthetic data with varying period value(p)

Tables 2, 3, and 4 represent the correlation values for synthetic data with varying period. The first row of the correlation tables represent the Pearson's coefficient and second row represents the Kendall Tau. It can be observed that, period value has a positive correlation w.r.t to the size of CSTs and negative correlation w.r.t to the size of the ST. The period value has positive correlation w.r.t the construction time in the case of CST_SCT3 and negative correlation in the case of CST_SADA and ST. The period value has positive correlation w.r.t the traversal time in the case of CST_SADA and negative correlation in the case of CST_SCT3 and ST.

Figure 1 and Table 5 show that, in general, with increasing period value the size of suffix tree and CST_SADA remain constant, whereas the size of CST_SCT3 increases slightly.



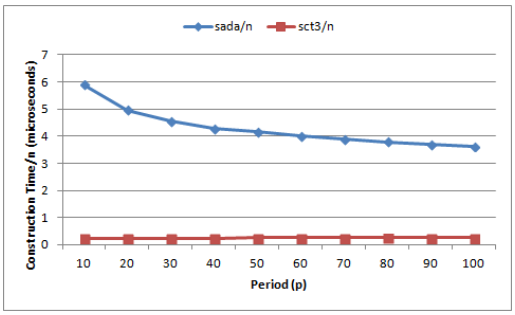
(a) ST (b) CST

Figure 1: Size per symbol(bytes) of suffix data structures with varying period value(p) using synthetic data

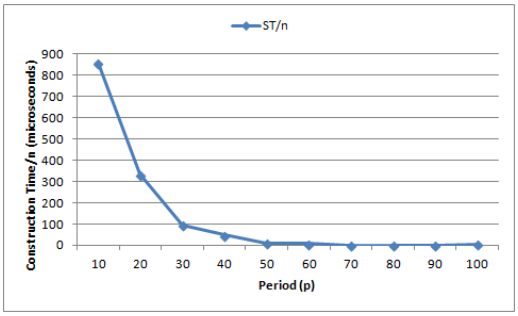
Figure 2 and Table 6 show that, in general, with increasing period value the time required to construct decreases in the case of CST_SADA and

slightly increases in the case of CST_SCT3. In the case of ST the construction time decreases rapidly with increasing period value.

Figure 3 and Table 7 show that, in general, with varying period value, there is no reasonable trend in the traversal time required for CSTs and ST.

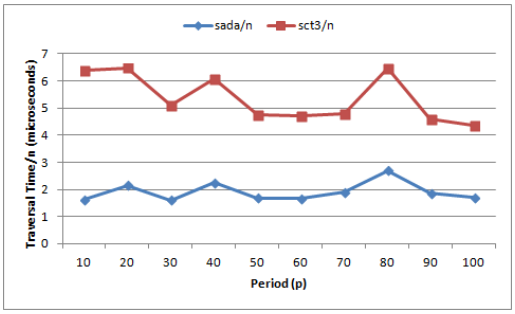


(a) CSTs

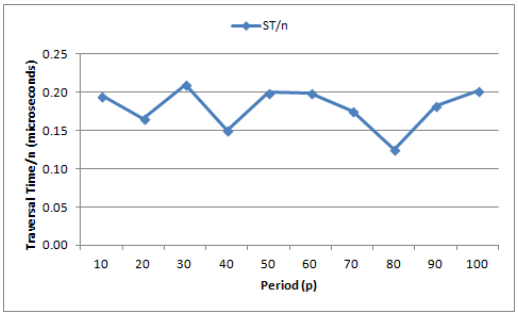


(b) ST

Figure 2: Construction time per symbol(μsecs) for suffix data structures with varying period(p) using synthetic data



(a) CSTs



(b) ST

Figure 3: Traversal time per symbol(μsecs) for suffix data structures with varying period(p) using synthetic data

p	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
10	1.267	4.144	145.000	1.267	4.144	145.000	114.459	34.986	3.272
20	1.267	4.182	144.999	1.267	4.182	144.999	114.465	34.675	3.301
30	1.267	4.197	144.999	1.267	4.197	144.999	114.427	34.544	3.312
40	1.267	4.199	144.998	1.267	4.199	144.998	114.418	34.535	3.313
50	1.268	4.208	144.998	1.268	4.208	144.998	114.389	34.454	3.320
60	1.267	4.220	144.998	1.267	4.220	144.998	114.407	34.362	3.330
70	1.268	4.218	144.997	1.268	4.218	144.997	114.384	34.380	3.327
80	1.268	4.214	144.997	1.268	4.214	144.997	114.329	34.409	3.323
90	1.268	4.220	144.996	1.268	4.220	144.996	114.338	34.360	3.328
100	1.269	4.216	144.996	1.269	4.216	144.996	114.281	34.391	3.323

Table 5: Size(MB) of suffix data structures with varying period value(p) using synthetic data

p	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
10	6.189	0.255	897.600	5.902	0.243	856.018	0.007	0.000	24.288
20	5.209	0.257	349.920	4.968	0.245	333.710	0.015	0.001	20.286
30	4.779	0.255	99.609	4.558	0.243	94.994	0.048	0.003	18.741
40	4.506	0.257	50.367	4.297	0.245	48.034	0.089	0.005	17.514
50	4.354	0.262	10.291	4.153	0.249	9.814	0.423	0.025	16.646
60	4.213	0.262	8.614	4.017	0.250	8.215	0.489	0.030	16.066
70	4.094	0.263	0.349	3.904	0.251	0.332	11.747	0.755	15.557
80	3.980	0.266	0.348	3.796	0.254	0.332	11.440	0.764	14.971
90	3.897	0.264	0.351	3.716	0.252	0.335	11.092	0.751	14.770
100	3.805	0.264	5.346	3.629	0.252	5.099	0.712	0.049	14.389

Table 6: Construction time(secs) for suffix data structures with varying period value(p) using synthetic data

p	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
10	1.699	6.697	0.332	1.620	6.387	0.196	5.111	20.149	3.942
20	2.248	6.802	0.373	2.144	6.487	0.166	6.028	18.243	3.026
30	1.689	5.341	0.355	1.611	5.094	0.210	4.758	15.049	3.163
40	2.360	6.382	0.355	2.251	6.087	0.151	6.643	17.967	2.705
50	1.756	4.992	0.352	1.675	4.761	0.200	4.994	14.192	2.842
60	1.752	4.958	0.350	1.671	4.728	0.200	5.011	14.176	2.829
70	1.987	5.006	0.349	1.895	4.774	0.176	5.696	14.349	2.519
80	2.819	6.791	0.354	2.689	6.476	0.126	7.964	19.183	2.409
90	1.953	4.805	0.356	1.862	4.582	0.182	5.484	13.495	2.461
100	1.784	4.558	0.360	1.701	4.347	0.202	4.949	12.647	2.555

Table 7: Traversal time(secs) for suffix data structures with varying period value(p) using synthetic data.

C Varying Alphabet Size ($|\Sigma|$)

Table 8 shows the LCP values and entropy information for synthetic data with varying alphabet size($|\Sigma|$). The LCP values remain constant for varying alphabet size, provided the period value($p = 32$) and data size($n = 10$ MB) remain constant. The H_0 and H_1 entropy values does not show any reasonable behavior with varying alphabet size, but they follow similar trends.

$\|\Sigma\|$	#numNodes	#numNodes/n	maxLCP	meanLCP	maxLCP/n	meanLCP/n	H_0	H_1
4	20,971,137	1.999963	10484760	5240000	0.999905	0.499725	1.999	1.993
8	20,970,979	1.999948	10484760	5240000	0.999905	0.499725	2.995	2.964
16	20,970,865	1.999938	10484760	5240000	0.999905	0.499725	3.988	3.816
32	20,970,820	1.999933	10484760	5240000	0.999905	0.499725	4.977	4.187
64	20,970,859	1.999937	10484760	5240000	0.999905	0.499725	3.468	2.904
128	20,970,777	1.999929	10484760	5240000	0.999905	0.499725	5.660	3.362
256	20,970,894	1.999940	10484760	5240000	0.999905	0.499725	4.536	2.901
512	20,970,863	1.999937	10484760	5240000	0.999905	0.499725	2.647	2.037

Table 8: LCP, entropy and other attributes for synthetic data with varying alphabet size($|\Sigma|$)

CST_SADA								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	-0.092	-0.473	-	-	0.307	-0.945	0.152	-0.296
$\|\Sigma\|$	0.286	-0.143	-	-	0.571	-0.909	0.357	-0.473

Table 9: Correlation values for CST_SADA using synthetic data with varying alphabet size($|\Sigma|$)

CST_SCT3								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	-0.092	-0.473	-	-	-0.089	-0.908	-0.075	-0.296
$\|\Sigma\|$	0.286	-0.143	-	-	0.286	-0.416	0.286	-0.473

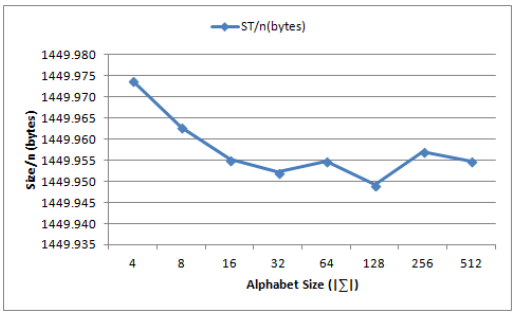
Table 10: Correlation values for CST_SCT3 using synthetic data with varying alphabet size($|\Sigma|$)

ST								
	H_0	H_1	maxLCP	meanLCP	size	cTime	tTime	# nodes
$\|\Sigma\|$	-0.092	-0.473	-	-	-0.286	0.954	-0.942	-0.296
$\|\Sigma\|$	0.286	-0.143	-	-	-0.416	0.929	-0.643	-0.473

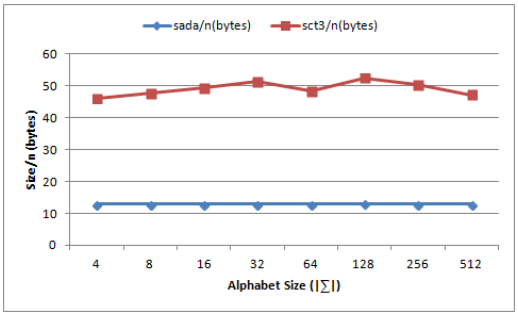
Table 11: Correlation values for ST using synthetic data with varying alphabet size($|\Sigma|$)

Tables 9, 10, and 11 represent the correlation values for synthetic data with varying alphabet. The first row of the correlation tables represent the Pearson's coefficient and second row represents the Kendall Tau. There is no variation in the LCP values for different alphabet size and hence there is no correlation between the alphabet size and the correlation values. It can be observed that, the alphabet size has no affect on the size of CST_SCT3, positive correlation w.r.t the size of CST_SADA and negative correlation w.r.t the size of ST. The alphabet size has a negative correlation w.r.t the construction time of CSTs and positive correlation w.r.t the construction time of ST. The alphabet size has a positive correlation w.r.t the traversal time of CST_SADA, negative correlation w.r.t the traversal time of ST and no affect on the traversal time of CST_SCT3.

Figure 4 and Table 12 show that, in general, with increasing alphabet size the size of suffix tree decreases, but does not have any reasonable affect on the size of CSTs.



(a) ST

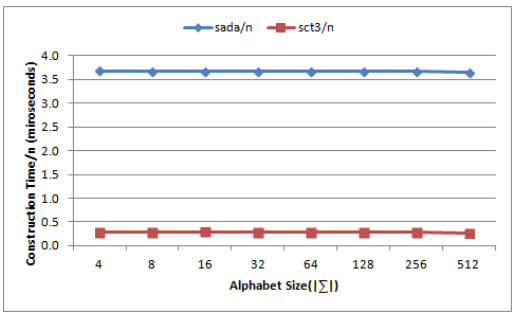


(b) CST

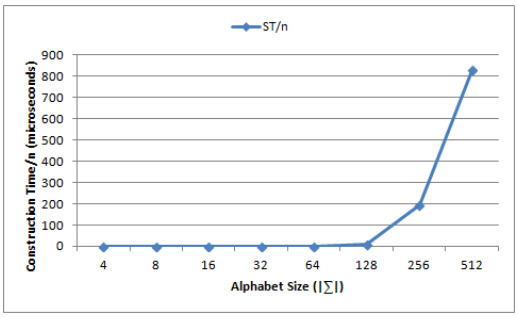
Figure 4: Size per symbol(bytes) of suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data

Figure 5 and Table 13 show that, in general, with increasing alphabet size the time required to construct the CSTs remain fairly constant, whereas in the case of ST it increases rapidly for larger alphabet sizes.

Figure 6 and Table 14 show that, in general, with varying alphabet size, the time required to traverse the ST and CST_SADA remains constant. For the case of CST_SCT3 the traversal time does not vary in linear w.r.t the varying alphabet size.

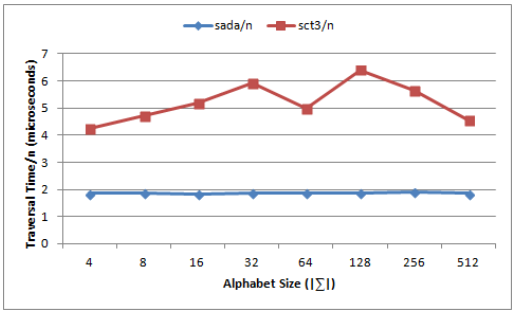


(a) CSTs

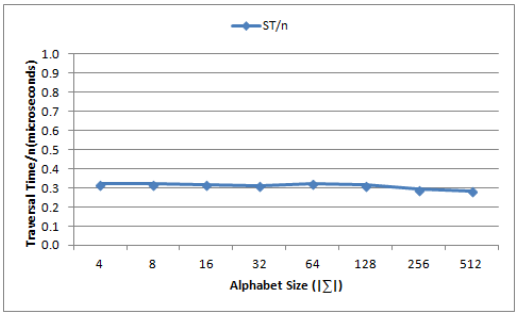


(b) ST

Figure 5: Construction time per symbol(μsecs) for suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data



(a) CSTs



(b) ST

Figure 6: Traversal time per symbol(μsecs) for suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data

$\|\Sigma\|$	sada(MB)	sct3(MB)	ST(MB)	sada/n(bytes)	sct3/n(bytes)	ST/n(bytes)	ST/sada	ST/sct3	sct3/sada
4	12.8830	46.176	1449.974	12.8830	46.176	1449.974	112.549	31.401	3.584
8	12.8834	47.798	1449.963	12.8834	47.798	1449.963	112.545	30.335	3.710
16	12.8838	49.535	1449.955	12.8838	49.535	1449.955	112.541	29.271	3.845
32	12.8843	51.334	1449.952	12.8843	51.334	1449.952	112.537	28.245	3.984
64	12.8835	48.499	1449.955	12.8835	48.499	1449.955	112.544	29.896	3.764
128	12.8851	52.574	1449.949	12.8851	52.574	1449.949	112.529	27.579	4.080
256	12.8850	50.484	1449.957	12.8850	50.484	1449.957	112.531	28.721	3.918
512	12.8839	47.248	1449.955	12.8839	47.248	1449.955	112.540	30.688	3.667

Table 12: Size(MB) of suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data

$\|\Sigma\|$	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sada/sct3
4	38.598	3.000	3.392	3.681	0.286	0.323	11.380	0.884	12.866
8	38.587	3.017	3.613	3.680	0.288	0.345	10.679	0.835	12.790
16	38.567	3.029	3.857	3.678	0.289	0.368	10.000	0.785	12.734
32	38.560	3.025	4.275	3.677	0.288	0.408	9.020	0.708	12.747
64	38.476	2.913	3.936	3.669	0.278	0.375	9.776	0.740	13.206
128	38.533	3.023	104.943	3.675	0.288	10.008	0.367	0.029	12.746
256	38.473	2.938	2052.440	3.669	0.280	195.736	0.019	0.001	13.097
512	38.280	2.701	8693.600	3.651	0.258	829.086	0.004	0.0003	14.175

Table 13: Construction time(secs) for suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data

$\|\Sigma\|$	sada(secs)	sct3(secs)	ST(secs)	sada/n(μsecs)	sct3/n(μsecs)	ST/n(μsecs)	sada/ST	sct3/ST	sct3/sada
4	19.392	44.401	3.363	1.849	4.234	0.321	5.766	13.203	2.290
8	19.584	49.494	3.355	1.868	4.720	0.320	5.837	14.752	2.527
16	19.097	54.258	3.343	1.821	5.174	0.319	5.712	16.229	2.841
32	19.523	61.992	3.266	1.862	5.912	0.311	5.978	18.984	3.175
64	19.739	52.432	3.373	1.882	5.000	0.322	5.851	15.542	2.656
128	19.674	67.191	3.292	1.876	6.408	0.314	5.976	20.409	3.415
256	19.835	59.295	3.067	1.892	5.655	0.293	6.467	19.332	2.989
512	19.416	47.671	2.967	1.852	4.546	0.283	6.545	16.069	2.455

Table 14: Traversal time(secs) for suffix data structures with varying alphabet size($|\Sigma|$) using synthetic data.